Universidad Carlos III de Madrid

Escuela Politécnica Superior

Ingeniería Informática (2º ciclo)

Proyecto Fin de Carrera

CONFLICTOS ENTRE ASPECTOS EN LA PROGRAMACIÓN ORIENTADA A ASPECTOS

Leganés, Madrid, España
Junio de 2009

Autor: Alicia Fernández Egido

Tutor: Eduardo Barra Zavaleta

ÍNDICE

1 INTRODUCCIÓN

1.1 Contexto

Los diferentes paradigmas de programación, funcional, procedural y en especial, la programación orientada a objetos, ofrecen potentes mecanismos para la separación de intereses o concerns (separation of concerns). Sin embargo, no ofrecen buenos resultados al tratar concerns que atraviesan todo o parte de un sistema. A estos concerns se les denomina crosscutting concerns.

Para resolver estas deficiencias surgió un nuevo paradigma de programación llamado programación orientada a aspectos (POA). Este nuevo paradigma encapsula los crosscutting concerns en unas nuevas unidades denominadas aspects, proporcionando una clara separación de concerns.

La POA permite tener un código no disperso y que las implementaciones sean más comprensibles, adaptables y reusables. Pero también tiene ciertos inconvenientes que hay que tener en cuenta para no obtener resultados inesperados y no deseados.

En las implementaciones que utilizan la POA pueden existir conflictos entre los distintos aspects de un sistema que pueden provocar comportamientos del sistema inadecuados.

1.1.1 Cronología de la Programación Orientada a Aspectos

70s David Parnas escribe "On the criteria to be used in decomposing systems into modules", en el que identifica la necesidad de crear módulos, bien sea por datos o por funcionalidad. Se muestran las ventajas de módulos por datos (origen de la programación orientada a objetos).

84 Concepto relacionado: Meta-programación. En meta-programación un programa puede manipularse a sí mismo, y mediante instrucciones comprender sus propias estructuras. Ejemplos: Smalltalk, reflexión en Java, Ruby.

92 Concepto relacionado: Filtros de composición.

95 Primeras referencias al término Programación por aspectos. Varios miembros del grupo de investigación de Demeter [6] escribieron un informe técnico sobre Separación de Concerns, identificando ciertas técnicas para manejar los concerns dispersos en un sistema.

95-97 Desarrollo de lenguajes de propósito específico con conceptos de aspectos (RG, AML, ETCML, DJ). La programación orientada a aspectos nació en los laboratorios de PARC [5] en un intento por conseguir estructuras de diseño complejas en implementaciones software. El trabajó se basó inicialmente en programación orientada a objetos, reflexión y protocolos de metaobjetos. Luego el proyecto se enfocó en los crosscutting concerns y se desarrollaron varios lenguajes de programación orientada a aspectos de propósito específico. El grupo de PARC estuvo liderado por Gregor Kickzales.

96 Primer workshop sobre el programación orientada a aspectos

98 Primera versión del lenguaje AspectJ. Se centró la atención en el desarrollo de un lenguaje de propósito general, y así nació AspectJ.

1.1.2 Objetivo

El objetivo de este proyecto es realizar un estudio de este nuevo paradigma de programación, y en especial de los conflictos entre aspects. Además, se abordará como caso práctico el desarrollo de un Detector de Conflictos que ayude a localizar y solucionar conflictos entre aspects. Para desarrollar el caso práctico se utilizará como lenguaje orientado a aspectos AspectJ, el más ampliamente utilizado.

1.1.3 Organización del documento

El documento se ha estructurado en seis puntos y un conjunto de anexos, tal y como se especifica a continuación:

Apartado 1. INTRODUCCIÓN. Se ofrece una introducción general al proyecto, exponiendo un nuevo paradigma, la POA y los conflictos como uno de sus inconvenientes, que se tratará de mejorar con el detector desarrollado en el proyecto.

Apartado 2. ESTADO DE LA CUESTIÓN. Se introducen conceptos necesarios para entender el paradigma de la POA. Además, se hace una introducción al lenguaje AspectJ, el más utilizado dentro del paradigma.

Apartado 3. DESARROLLO TEÓRICO. En este apartado se explica con detalle lo que se entiende por conflicto entre aspects, los distintos tipos de conflictos y las distintas clasificaciones que existen sobre conflictos. También se muestra como trata los conflictos el lenguaje AspectJ y las herramientas existentes que abordan los conflictos.

Apartado 4. EXPERIMENTACIÓN. Es la parte práctica del proyecto, en la cual se detalla la propuesta del detector de conflictos desarrollado. En este apartado también se muestra el proyecto Telecom como caso práctico para mostrar el funcionamiento del detector.

Apartado 5. CONCLUSIONES. Se analizan las ventajas de la POA, así como sus inconvenientes. Uno de ellos, los conflictos entre aspects

Apartado 6. TRABAJOS FUTUROS. Se muestran las posibles mejoras al proyecto desarrollado, así como futuras líneas a seguir.

Apartado 7. ANEXOS. Como anexo se muestra cómo instalar AspectJ en Eclipse, cómo crear un proyecto, y cómo compilar y ejecutar un proyecto que combina aspects y clases en Eclipse.

Apartado 8. REFERENCIAS. Se presenta la documentación, así como direcciones de Internet, utilizadas para la realización del proyecto.

2 ESTADO DE LA CUESTIÓN

2.1 PROGRAMACIÓN ORIENTADA A ASPECTOS

La historia del software ha ido evolucionando desde paradigmas de programación orientados a la máquina, como código ensamblador, a los más recientes, como programación orientada a objetos, los cuales tratan de simular el modo de pensar del hombre.

La ingeniería del software siempre ha buscado mejorar la calidad del software [1], reducir costes de producción del software y facilitar el mantenimiento y la evolución. Y para alcanzar estos objetivos constantemente se buscan nuevas metodologías y tecnologías que reduzcan la complejidad del software, promuevan la reusabilidad y faciliten la evolución.

Uno de los avances más importantes en los últimos tiempos en la ingeniería del software ha sido la programación orientada a objetos, la cual utiliza abstracciones que representan entidades del dominio del problema. Estas abstracciones son los objetos, y engloban entidades del dominio con propiedades y comportamientos similares.

La programación orientada a objetos busca conseguir un software de calidad a través de la descomposición, la abstracción, la encapsulación y la ocultación de la información. Pero el modelo de objetos tiene limitaciones que hacen que el software se vaya degradando con el tiempo, y que las actividades de mantenimiento y evolución sean complicadas, costosas y tengan gran impacto en el sistema.

La descomposición del sistema en partes más fáciles de manejar no plantea problemas para descomponer la funcionalidad básica del dominio, pero no responde adecuadamente a otra funcionalidad [2]. La descomposición utilizada en programación orientada a objetos se basa en dividir el sistema en varios subsistemas que corresponden a diferentes partes del dominio, pero que sin embargo, no se adapta bien a ciertos requerimientos, especialmente los no funcionales, como manejo de errores, logging, sincronización, etc.

Así, si queremos añadir funcionalidad de logging a nuestro sistema, la solución suele consistir en repetir código que resuelva este requerimiento en los distintos subsistemas que lo necesitan. Esto es lo que se conoce como crosscutting concern, partes del sistema que no pueden ser encapsulados en una única entidad puesto que atraviesan múltiples partes del sistema. El resultado final es un código mezclado y diseminado (tangling and scattering code) [3], porque un mismo módulo recoge varios requerimientos, y un requerimiento se encuentra diseminado por varios módulos.

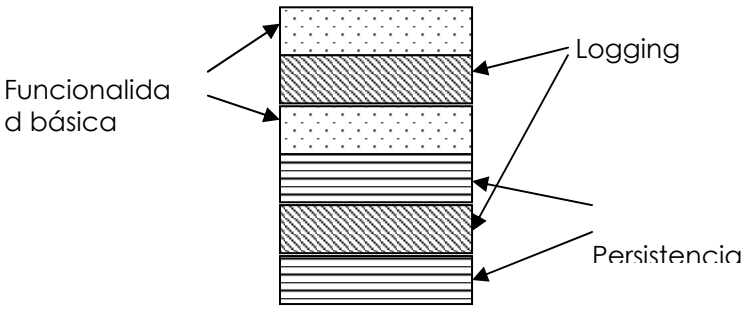


Figura 1. Módulo con código diseminado

Todo esto produce los siguientes efectos negativos sobre el sistema software [3]:

- Baja correspondencia entre un concepto y su implementación.
- Menor productividad de los desarrolladores que tienen que preocuparse de conceptos no relativos al dominio.
- Código poco reusable y de baja calidad. Se trata de un código propenso a errores, en el que futuros cambios resultarán costosos y complicados.

2.1.1 Separación de concerns

Lo expuesto en el apartado anterior nos lleva al problema más importante que nos encontramos en la programación orientada a objetos, la separación de concerns tanto en diseño como en implementación, entendiendo por concerns, aquellos temas o asuntos de los que es necesario ocuparse para resolver un problema. Así, entendemos por

Separación de concerns la habilidad para identificar, encapsular y manipular sólo las partes del software correspondientes a un concern.

El concern más importante es la función específica que debe realizar la aplicación, pero existen otros concerns como sincronización, seguridad, logging, persistencia, manejo de errores, etc. Estos últimos se denominan crosscutting concerns porque atraviesan el sistema y no permiten su modularización en programación orientada a objetos. Una clara separación de concerns hará que el sistema sea fácil de manejar, entender, mantener y evolucionar.

En general, los mecanismos existentes para la composición y descomposición del sistema soportan una "dimensión dominante", lo cual se conoce como la tiranía de la descomposición dominante (tyranny of the dominant decomposition) [1]. Esto es, un sistema consta de muchas dimensiones de concerns que facilitan la organización del sistema. Unas dimensiones incluyen conceptos funcionales propios del dominio, mientras que otros son conceptos cross-cutting, como el logging. Pero casi todos los formalismos utilizados en el desarrollo del software permiten la descomposición del sistema de acuerdo a una única dimensión de concern. Así, la programación orientada a objetos soporta descomposición basada en la dimensión del objeto. La clase modeliza un tipo de objeto y todos los detalles de los objetos están descritos dentro de la clase. Esto hace que el resto de dimensiones de concerns se tengan que mezclar con la dimensión dominante como muestra la Figura 2, afectando a la separación de concerns.

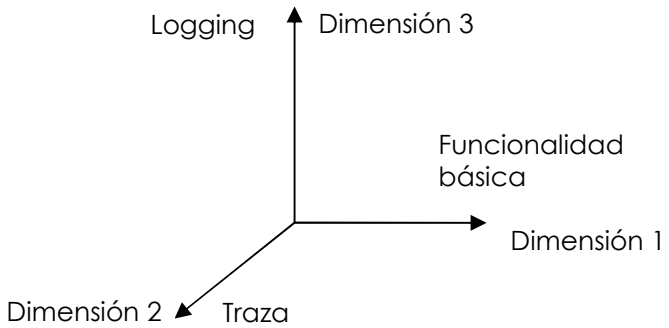


Figura 2. Varias dimensiones de concerns y una dimensión dominante

Todos los sistemas, por muy pequeños que sean, están compuestos de unidades elementales. En la tarea de desarrollo de un sistema, el ingeniero

de sistemas debe ser capaz de centrarse en esas unidades e ignorar el resto. Para esto, el programador identifica los concerns de importancia y encapsula las unidades pertenecientes a un concern en un módulo. Idealmente, si uno quisiera saber a cerca de ese concern debería buscar en el módulo correspondiente. En este contexto, la programación orientada a aspectos (POA) surge para resolver el problema de la separación de concerns, proponiendo modularizar los crosscutting concerns en unas nuevas unidades denominadas aspectos (aspects).

2.1.2 Fundamentos de la Programación Orientada a Aspectos

El término POA (Programación Orientada a Aspectos) describe una forma de programación basada en conceptos que generalmente no forman parte de la funcionalidad básica, pero que cruzan todo o parte del sistema. La POA permite definir estos conceptos, llamados aspectos, en entidades separadas, y en un momento dado juntarlos de forma adecuada con el resto de funcionalidad básica para formar el sistema final.

Según Kiczales:

"Un aspecto es una unidad modular que se encuentra dispersa por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la de implementación.

Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño.

Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades modulares del programa".

De las definiciones anteriores podemos decir que, en el nivel de diseño, un aspect es una unidad que se dispersa en el sistema. Mientras que en el nivel de implementación, un aspect es un constructor que nos permite modularizar un concern que se dispersa por el sistema.

Algunos ejemplos de estos aspects son manejo de errores, logging, sincronización o persistencia.

La POA tiene dos características que las diferencia de otros mecanismos: Inconciencia (Obliviousness) y Quantification [4].

Inconciencia se refiere al hecho de no saber qué aspecto se ejecutará observando el código base. Esto permite una mayor separación de concerns ya que esta separación no solo existe a nivel de implementación sino en la mente de los desarrolladores.

Cuantificación es la propiedad que permite hacer afirmaciones del tipo: cada vez que se ejecute la condición C sobre el objeto O, realiza la acción A. Además, por la propiedad anterior de Inconciencia, esta afirmación se podrá implementar en un módulo separado del código base.

2.1.3 Conceptos de la Programación Orientada a Aspectos

La siguiente figura muestra los puntos principales en un desarrollo de software orientado a aspectos [7]:

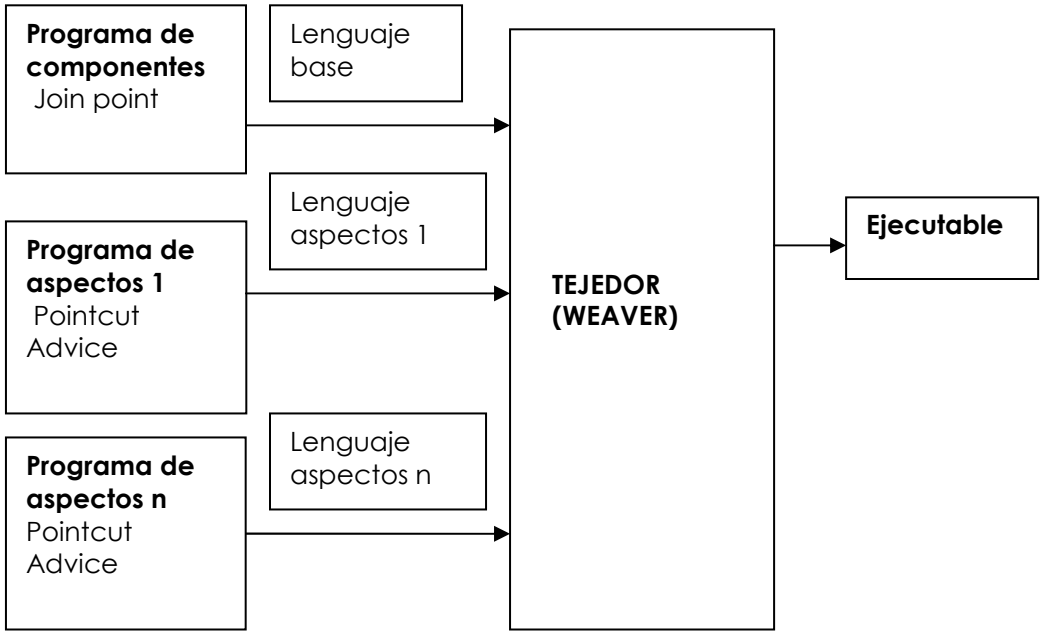


Figura 3. Elementos principales de la programación orientada a aspectos

Los tres elementos principales de la POA son:

- Un lenguaje para definir la funcionalidad básica, conocido como lenguaje base o de componentes. Ejemplos de estos lenguajes son Java, C, C++.

- Uno o varios lenguajes de aspectos para especificar el comportamiento de los diferentes aspectos. Ejemplos de este tipo de lenguajes son COOL, RIDL o AspectJ.

- Un tejedor de aspector o weaver que se encarga de combinar los lenguajes base y de aspectos en tiempo de ejecución o compilación.

A continuación, vamos a ir viendo cada uno de los elementos del esquema y otros que no se encuentran en él, pero que son necesarios para entender esta forma de programación.

Para la mayoría de términos utilizaremos en el resto del documento la versión en inglés, ya que la traducción al español no siempre es la misma y puede dar lugar a confusión porque en distintos textos se utilizan distintas traducciones.

Crosscutting concerns (Conceptos entrecruzados)

Decimos que un concepto se encuentra disperso (crosscutting concern) en un sistema cuando la implementación de un determinado concepto se reparte a lo largo de toda o parte de la implementación.

En programación orientada a aspectos el concepto sigue estando disperso a lo largo del sistema, pero en lo relativo a la implementación, el código relativo al aspecto se localiza en un módulo separado de donde se necesita.

Aspects (Aspectos)

Un aspect es la unidad básica en la programación orientada a aspectos para implementar un crosscutting concern. Es muy parecido a una clase de programación orientada a objetos, con un nombre, unas variables y unos métodos.

La definición de un aspect contiene código (advice) e instrucciones acerca de dónde, cuándo y cómo invocarlo.

Joinpoints (Puntos de unión)

Los joinpoints son puntos bien definidos en la estructura de un programa que permiten añadir funcionalidad adicional. Los más comunes son llamadas a procedimientos, aunque algunos lenguajes permiten que también sean puntos de unión definiciones de campos, modificación de campos, acceso a campos o excepciones.

Pointcuts (Puntos de corte)

Un pointcut describe un conjunto de joinpoints. Se trata de una herramienta muy útil para referenciar todos los joinpoints donde cierto código se quiere invocar, reduciendo el riesgo de invocar incorrectamente un aspecto.

Mientras que los joinpoints hacen referencia a un concepto, los pointcut son nuevos constructores.

Advice (Avisos)

Un advice corresponde a las acciones que se ejecutan en cada joinpoint incluido en un pointcut.

Un advice es transparente al programador del código base, puesto que al contrario que en los lenguajes de programación convencional, las acciones que se ejecutarán al alcanzar los joinpoints no están explícitas en el código base.

Muchos lenguajes de aspectos permiten ejecutar estos advice antes, después, en vez de o alrededor del joinpoint al que hacen referencia.

Lenguajes de aspectos

Los lenguajes de aspectos hacen referencia a los lenguajes de programación utilizados para implementar los aspects. No se utilizan por sí solos, sino como extensión a los lenguajes base para dar soporte a los aspects.

En el siguiente capítulo se habla más detenidamente sobre los lenguajes de aspectos.

Lenguaje base

El lenguaje base se refiere al lenguaje de programación utilizado para implementar los componentes que contienen la funcionalidad principal del sistema.

A la hora de elegir un lenguaje base, habrá que decidir si se adopta uno ya existente o se diseña un nuevo lenguaje. Hasta el momento, se está optanto por utilizar un lenguaje base ya existente por las ventajas que supone, los desarrolladores trabajan con un lenguaje conocido y altamente probado. El problema puede venir si se considera que el lenguaje base interfiere para una completa separación de concerns (principal objetivo de la POA).

Ejemplos de lenguaje base utilizados en programación orientada a aspectos son Java, C, C-Sharp.

Tejedor (weaver)

Es el mecanismo encargado de componer los componentes funcionales del sistema con los aspects y formar el sistema final, bien en tiempo de compilación o en tiempo de ejecución.

Para combinar o "tejer" los componentes y los aspects, el lenguaje base y el lenguaje de aspectos tienen que interaccionar haciendo uso de los puntos de unión y añadiendo el comportamiento adicional contenido en los aspects.

Existen dos formas de enlazar el código base con el código de aspectos:

- Tejido estático: consiste en modificar el código fuente del lenguaje base, insertando el nuevo código dónde indican los puntos de enlace. Esto es, el código de aspectos se introduce en el código fuente principal antes de la compilación. Un ejemplo de este tipo de tejido lo encontramos en AspectJ. Los aspects están fuertemente relacionados con las clases y el código está altamente optimizado (la velocidad de ejecución es comparable a la del código sin aspects).

- Tejido dinámico: este tipo de tejido requiere que los aspects existan y estén presentes tanto en tiempo de compilación como en tiempo de ejecución. En este tipo, el tejedor será capaz de añadir, adaptar y eliminar aspectos de forma dinámica durante la ejecución según las necesidades.

Este tipo se utiliza en el tejedor de Jboss AOP. Los aspects y las clases están debilmente relacionados y resulta ineficiente ya que se necesita código extra para acoplar y desacoplar los aspects y las clases en tiempo de ejecución.

2.2 LENGUAJES DE ASPECTOS

2.2.1 Tipos de lenguajes de aspectos

Existen dos enfoques principalmente:

- Lenguajes de aspectos de dominio específico: capaces de manejar uno o más aspects, pero unicamente esos aspects. Tienen un gran nivel de abstracción, y para garantizar que los aspectos del dominio son escritos en el lenguaje de aspectos, se utiliza un lenguaje base restringido.

Ejemplos de lenguajes de aspectos específicos son [7]:

- COOL (COOrdination Language) de XEROX . Se trata de un lenguaje para especificar sincronización de hilos concurrentes, donde el lenguaje base es una versión de Java restringido en el cual se eliminan los métodos wait,notify y notifyAll, y la palabra reservada syncronized, evitando que el lenguaje base pueda expresar sincronización.
- RIDL (Remote Interaction and Data transfers aspect Language). Este lenguaje maneja la transferencia de datos entre diferentes espacios de ejecución. Un programa RIDL consite en un conjunto de módulos de portales o simplemente portales, los cuales se asocian con las clases por el nombre. Un portal se encarga de manejar la interacción remota y la transferencia de datos de la clase asociada a él, pudiendo asociarse como máximo a una clase, y siendo la unidad mínima de interacción remota el método.
- MALAJ es otro lenguaje de aspectos específico que modulariza los aspects de sincronización y relocación. Se puede ver como el sucesor de COOL y RIDL e impone una mayor restricción sobre la visibilidad de los aspects. Por cada aspect, provee un constructor lingüístico distinto, limitando la visibilidad del aspect sobre el módulo asociado.

- Lenguajes de aspectos de propósito general: permiten describir cualquier tipo de aspect, no solo específicos. El nivel de abstracción del lenguaje base y del lenguaje de aspectos es el mismo, y comparten el mismo conjunto de instrucciones. Sin embargo, no garantizan la completa separación de concerns ya que, al no imponer restricciones al lenguaje base, no garantizan que no se programe el aspect unicamente en el lenguaje de aspectos. Estos lenguajes tienen como ventaja que los desarrolladores no tienen que cambiar de lenguaje para programar distintos tipos de aspects y tampoco tienen que aprender distintos lenguajes. Ejemplos de lenguajes de propósito general son:

    - AspectJ [15], lenguaje de aspectos de propósito general que extiende Java para dar soporte al manejo de aspects, añadiendo a la semántica de Java cuatro entidades principalmente: puntos de enlace o joinpoints, cortes o pointcuts, avisos o advices e introducciones o introductions. Actualmente es el lenguaje más utilizado y lo veremos en el siguiente apartado más extensamente.

    - AspectC++ [16], lenguaje que extiende al lenguaje C++ para definir los aspects, basado en el modelo del lenguaje AspectJ. Como en AspectJ, los conceptos más importantes son joinpoint, pointcut y advice. El weaver de AspectC++ se conoce como ac++ y transforma los programas AspectC++ en programas C++.

    - Eos [17] es el lenguaje de aspectos que amplía el lenguaje de programación C# para el entorno .NET de Microsoft. Soporta la modularización de los crosscutting concerns mediante tres nuevos constructores: pointcuts, bindings y declaraciones inter-type. El pointcut captura puntos en la ejecución del programa llamados joinpoints, mientras que el binding conecta los joinpoints con métodos. El constructor binding utiliza el constructor pointcut para seleccionar un grupo de joinpoints y asociarlos a un método. La declaración inter-type permite añadir miembros e interfaces adicionales a un tipo sin involucrar al tipo en sí.

    - phpAspect [18] es una extensión al lenguaje PHP que implementa programación orientada a objetos para PHP 5. El

proceso de weaving es estático y basado en el análisis de Lex y Yacc para generar árboles de parseado en xml. XSLT es utilizado para generar estos árboles a partir del código fuente. El objetivo de phpAspect es ofrecer una extensión al lenguaje PHP con el que implementar los concerns específicos de web.

2.2.2 ASPECTJ

Se ha elegido AspectJ para profundizar en un lenguaje de aspectos y utilizarlo en la parte práctica porque sin duda es el lenguaje más ampliamente utilizado y más documentado.

Este apartado dedicado al lenguaje AspectJ tiene como objetivo dar a conocer el lenguaje de aspectos AspectJ y ser una referencia para entender el código AspectJ incluido en este proyecto.

Se tomó como referencia el libro 'AspectJ Cookbook' [8].

El lenguaje AspectJ es un lenguaje de propósito general que extiende al lenguaje Java para el soporte de aspects. Fue desarrollado en 1998 por Gregor Kickzales y el grupo de investigación dirigido por PARC (Palo Alto Research Center), una subsidiaria de Xerox. En diciembre del 2002 el proyecto AspectJ fue cedido a la comunidad open-source Eclispse.org, la cual continúa mejorando y dando soporte al proyecto.

Al ser una extensión del lenguaje Java es fácil de adoptar por programadores Java. Además, cualquier programa correcto en Java será un programa correcto en AspectJ, y todo programa correcto en AspectJ podrá correr sobre una máquina virtual de Java.

2.2.2.1 AspectJ añade a Java

Constructores que afectan dinámicamente al flujo del programa:

- Joinpoints. Se trata de puntos bien definidos en la ejecución de un programa (no son posiciones en el código fuente). Ejemplos de joinpoints son llamadas a método, acceso a atributos, etc. Un mayor número de tipos de joinpoints significará mayor número de casos a representar en un aspect, y por lo tanto, más potente será la herramienta.
- Pointcuts: son estructuras lógicas que permiten definir joinpoints dentro de un aspect. Los pointcuts son constructores

que agrupan joinpoints y datos del contexto de la ejecución de dichos joinpoints. Por ejemplo, mediante un pointcut se pueden agrupar todas las llamadas a un método de una clase y acceder a su contexto (argumentos, objeto invocador, objeto receptor).

- Advice: especifican el código que será ejecutado al alcanzar los joinpoint que satisfacen ciertos pointcuts. Son equivalentes a los métodos y añaden el comportamiento transversal que contendrá el aspect. Además de indicar dónde se inserta este código, un advice indica la forma de insertarlo (antes, después o reemplazando el pointcut al que hace referencia).

Constructores que modifican de forma estática la jerarquía de clases del programa:

- Declaraciones inter-tipo (Inter-type declarations): son declaraciones que permiten añadir métodos y atributos a una clase o interface existente.
- Declaraciones de parentesco (declare parents): se utiza para especificar que una clase determinada hereda de otra.
- Declaraciones de precedencia (declare precedence): especifican el orden de aplicación de distintos aspects que se encuentran en conflicto al afectar al mismo joinpoint.
- Declaraciones en tiempo de compilación (compile-time declarations): pueden ser de tipo warning o de tipo error, y generan un aviso o un error en tiempo de compilación si se cumple una determinada condición.
- Excepciones suavizadas (softening exceptions): elimina la obligación de capturar las excepciones.

2.2.2.2 Sintaxis y semántica de los conceptos de AspectJ

Aspect

- Aspects. Los constructores descritos anteriormente se encapsulan en el aspect. Un aspect es la unidad básica de AspectJ, así como un objeto es la unidad básica en Java.

Un aspect es la unidad de AspectJ que encapsula joinpoint, pointcuts, advices y declaraciones. Además, contiene sus propios atributos y métodos como una clase normal de Java.

El código del aspect puede situarse en un fichero aparte con extensión .aj o incluirse como un miembro más de la clase si el aspect solo afecta a una clase dentro de la aplicación.

Los aspects se comportan de forma parecida a las clases:

- Pueden contener métodos, variables de instancia, etc.
- Pueden ser declarados public, private o final.
- Pueden ser abstractos, lo que les permite una mayor reusabilidad.
- Los pointcuts también pueden ser abstractos siempre que se declaren en aspects abstractos.
- Pueden extender clases, otros aspects o implementar interfaces.
- Sin embargo, existen diferencias entre aspects y clases:
- Los aspects no poseen constructores, por lo tanto no pueden ser instanciados como una clase normal Java. La VM se encarga de ello, creando la instancia del aspecto de forma automática según el tipo de instanciación del aspect que puede ser: singleton, per-object o per-control-flow (la instanciación por defecto es singleton).
- No se pueden sobrecargar.
- Un aspect no puede heredar de un aspect concreto.
- Pueden ser marcados como privilegiados (privileged), lo que significa que podrían acceder a miembros privados de las clases. Se debe utilizar en situaciones especiales porque viola el principio de encapsulación.

Extensión de Aspects

Un aspect puede extenderse de las siguientes formas:

- Mediante un aspect abstracto con el modificador abstract: este aspect abstracto se usará como base para otros aspects. Un aspect abstracto no puede heredar aspects concretos.
- Cuando un aspect hereda un aspect abstracto además de heredar los atributos y los métodos, también hereda los joinpoints y los advice.

- Heredando de clases o implementando interfaces.

Precedencia de aspects

Si distintos advice en distintos aspects se corresponden con un mismo joinpoint se puede utilizar la sentencia precedence para especificar relaciones de precedencia entre aspects.

Se puede utilizar caracteres comodín para especificar los tipos.

Sintaxis:

declare precedence : <Tipo>, <Tipo>,…;

Ejemplos:

declare precedence: AspectA, AspectB;	aspectA se ejecutará antes que AspectB
declare precedence : AspectA, *;	aspectA se ejecutará antes que el resto
declare precedence : AspectA+,* ;	Se ejecutarán primero los aspectos que extienden el aspecto AspectA

Tabla 1. Ejemplos precedencia en AspectJ.

Instanciación de aspects

Las instancias de un aspect se crean automáticamente según el tipo de instanciación del aspect, el cual puede ser: singleton, per-object y per-control-flow. Las características de cada uno de estos tipos de instanciación son:

- Singleton
    - Una única instancia será creada para toda la aplicación y estará disponible durante toda la ejecución de la aplicación a través del método estático del aspect aspectOf ( ).

- Sintaxis: aspect <nombre Aspecto> issingleton ( ) {
  .....
  }

- Per-object

Existen dos variantes de este tipo:

- per-this: se crea una instancia del aspect por cada objeto actual (this) distinto asociado a los joinpoints correspondientes a un pointcut. Para obtener la instancia del aspecto habrá que pasar al método aspectOf ( ) el objeto actual: aspectOf (thisJoinPoint.getThis( )).
  - Sintaxis: aspect <nombre Aspecto> perthis (<pointcut>) {
    .....
    }
- per-target: se crea una instancia del aspect por cada objeto destino (target) distinto asociado a los joinpoints correspondientes a un pointcut.

  Para obtener la instancia del aspect habrá que pasar al método aspectOf ( ) el objeto destino: aspectOf (thisJoinPoint.getTarget( )).
  - Sintaxis: aspect <nombre Aspecto> pertarget (<pointcut>) {
    .....
    }

- Per-control-flow

Este tipo contiene dos variantes:

- per-cflow: se crea una instancia del aspect cada vez que se ejecuta un joinpoint bajo el flujo de control de los puntos de enlace identificados por un pointcut, incluyendo los propios joinpoints identificados por el pointcut.
  - Sintaxis: aspect <nombre Aspecto> percflow (<pointcut>) {

```
        .......
    }
```

- per-cflowbelow: se crea una instancia del aspect cada vez que se ejecuta un joinpoint bajo el flujo de control de los puntos de enlace identificados por un pointcut, excluyendo los propios joinpoints identificados por el pointcut.
    - Sintaxis: aspect <nombre Aspecto> percflow (<pointcut>) {
      ....
      }
- Para los dos variantes, para obtener la instancia del objeto en tiempo de ejecución usaremos el método del aspecto aspectOf( ). No hace falta pasarle ningún parámetro al método porque cada vez que se ejecuta una instancia del aspect, al llegar al joinpoint, la instancia se mete en una pila y luego, se destruye al terminar la ejecución del joinpoint. Así, al llamar al método aspectOf ( ), éste devolverá la instancia que hay en el tope de la pila.

Joinpoints

Los joinpoints son puntos bien definidos en la ejecución de un programa, dónde se insertará cierta funcionalidad incluida en los advice.

AspectJ puede detectar y operar sobre los siguientes once tipos de joinpoints:

- Llamada a método: cuando un método es invocado por un objeto.

Ejemplo:

```
clase.metodo(arg);          ← Joinpoint llamada a método
```

- Ejecución de método: engloba la ejecución del cuerpo de un método.

Ejemplo:

```
public void metodo (int x){
    this.x = x;                    ] Joinpoint ejecución método
    }
```

- Llamada a constructor: cuando un constructor es invocado en la creación de un nuevo objeto.

Ejemplo:

```
Clase c = new Clase (x);    ← Joinpoint llamada a constructor
```

- Ejecución de constructor: abarca la ejecución del cuerpo de un constructor durante la creación de un nuevo objeto.

Ejemplo:

```
public Clase (int x) {
   this.x = x;                    ] Joinpoint   ejecución de
constructor
}
```

- Inicialización de clase: cuando se ejecuta el inicializador estático de una clase, justo en el momento en el que el cargador de clases carga dicha clase.

Ejemplo:

```
public class Main {
   ……..
   static {
   …… ] Joinpoint inicialización de clase
   }
}
```

- Preinicialización de objeto: ocurre antes de que se ejecute la inicialización de una clase.

Ejemplo:

```
public Clase (int x, int y, int z) {
   super(x,
   ClasePadre.metodo (y)); ←Joinpoint
                                  preinicialización objeto
   );
   this.z=z;
}
```

- Inicialización de objeto: cuando un objeto es creado. Comprende desde el retorno del constructor padre hasta el final del primer constructor llamado.

Ejemplo:

```
public Clase (int x) {
```

```
        super ( );
        this.x = x;                    ← Joinpoint inicialización de objeto
    }
```

- Lectura de un atributo: cuando se lee un atributo de un objeto dentro de una expresión.

Ejemplo:

```
    public int getX ( ) {
        return x ;                     ← Joinpoint lectura del atributo x
    }
```

- Escritura de un atributo: cuando se asigna un valor a un atributo de un objeto dentro de una expresión.

Ejemplo:

```
    public int setX (int x) {
        x = i ;                        ← Joinpoint escritura del atributo x
    }
```

- Ejecución de manejador de excepciones: cuando un manejador de excepciones es ejecutado.

Ejemplo:

```
    try {
        ……..
    }catch (IOException e) {
        System.println("Manejador"); ]Joinpoint manejador
excepciones
    }
```

- Ejecución de advice: comprende la ejecuión de un advice.

Ejemplo:

```
    after ( ) : ejemploAdvice ( ) {
        ……                         ] Joinpoint ejecución de advice
    }
```

Pointcuts

Los pointcuts se utilizan para capturar los diferentes joinpoints.

Los pointcuts pueden tener nombre o ser anónimos (sin nombre). Los pointcut anónimos se definen donde se van a utilizar (en un advice o en

otro punto de corte) y no pueden ser utilizados en otra parte del código. Los puntos de corte con nombre si pueden reutilizarse en distintas partes del código.

La sintaxis de un pointcut anónimo es la siguiente:

<tipo de advice> (argumentos) : <expresion> [&& | ||]

donde tipo de advice indica si se trata de un advice before, after o around, y cada una de las expresiones está formada por un tipo de corte y una signatura. Se pueden enlazar varias expresiones utilizando los operadores lógicos AND y OR.

La sintaxis de un pointcut con nombre es:

pointcut <nombrePoincut> (<argumentos>): Cuerpo;

donde nombrePointcut es el nombre del pointcut que debe cumplir las normas de Java para identificadores, argumentos es la lista de argumentos que serán pasados a los advice correspondientes, y Cuerpo contiene la definición del pointcut, pudiendo ser pointcuts primitivos o una combinación de otros pointcut.

Los comodines y operadores que pueden utilizarse en los pointcut permiten seleccionar los joinpoint que va a capturar el pointcut.

* Sustituye a un conjunto de caracteres, excepto el punto.

.. Sustituye a un conjunto de parámetros.

\+ Busca los joinpoint dentro del package o subpackage.

! Operador NOT. Utilizado para no seleccionar el joinpoint indicado.

|| Operador OR. Se utiliza para unir un conjunto de pointcuts con la condición de uno u otro.

&& Operador AND. Se utiliza para unir un conjunto de pointcuts con la condición de uno y otro.

Los pointcut proporcionados por AspectJ se clasifican en diferentes grupos:

- Basados en las categorías de joinpoints: capturan los joinpoints según la categoría correspondiente: llamada a método, ejecución de método, etc.
- Basados en el flujo de control: capturan joinpoints de cualquier categoría siempre que ocurran en el contexto de otro pointcut. Estos pointcuts son: cflow y cflowbelow.
- Basados en la localización de código: capturan joinpoints de cualquier categoría que se localizan en ciertos fragmentos de código, por ejemplo, dentro de una clase o dentro del cuerpo de un método. Estos pointcuts son: within y withincode.
- Basados en los objetos en tiempos de ejecución: capturan los joinpoints cuyo objeto actual (this) u objeto destino (target) son de un cierto tipo. Además de capturar los joinpoints asociados con los objetos referenciados, permiten exponer el contexto de los joinpoints.
- Basados en los argumentos del joinpoint: capturan los joinpoints cuyos argumentos son de un determinado tipo mediante el descriptor args.
- Basados en condiciones: capturan los joinpoints que cumplen una determinada condición usando el descriptor if(expresionBooleana).

Los tipos de pointcuts definidos en AspectJ son:

- Llamada a método: Captura la llamada a un método de una clase.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : call (<modificador opcional> <tipo retorno> <clase>.<metodo>(<tipos parámetros>));

El pointcut call(Signatura) tiene dos características:

- Advice es capturado sobre la llamada a un método. Y el contexto es el de la clase que llama.
- La Signatura puede contener caracteres comodines para seleccionar un conjunto de joinpoints sobre diferentes clases y métodos.

Ejemplos:

Signatura	Descripción
* void MiClase.log (int, float) void MiClase.log (int, float)	Captura la llamada al método log dentro de la clase MiClase sin importar el modificador
* * MiClase.log(int, float) * MiClase.log(int, float)	Captura la llamada al método log dentro de la clase MiClase sin importar el modificador ni el tipo de retorno
* * *.log(int, float) * * log(int, float)	Captura la llamada al método log sin importar el modificador, ni el tipo de retorno ni la clase.
* * *.*(int, float)	Captura cualquier llamada a método sin importar el modificador, ni el tipo de retorno, ni la clase, ni el método.
* * *.*(*,float)	Captura cualquier llamada a método sin importar el modificador, ni el tipo de retorno, ni la clase, ni el método donde los parámetros incluyen lo que sea seguido de un float.
* * *.*(*,..)	Captura cualquier llamada a método sin importar el modificador, ni el tipo de retorno, ni la clase, ni el método donde los parámetros incluyen al menos un valor seguido de cualquier número de parámetros.

* * *.*(..) * *(..)	Captura cualquier llamada a método sin importar el modificador, ni el tipo de retorno, ni la clase, ni el método, incluyendo ninguno o algún parámetro.
* mipaquete..*.*(..)	Captura cualquier llamada a método dentro del paquete mipaquete y sus subpaquetes
* MiClase+.*(..)	Captura cualquier llamada a método que se encuentre en la clase MiClase y cualquier subclase de ésta.

Tabla 2. Ejemplos de Signatura en AspectJ

Se pueden capturar los valores de los parámetros pasados en una llamada a método. Para ello hay que especificar los parámetros como identificadores en la signatura y añadir el pointcut args(identificadores), para luego ligar los identificadores requeridos con los valores de los parámetros del método. A continuación se muestra un ejemplo:

```
pointcut capturaCallParametros(int valor, String nombre) :
        call(void MiClase.log(int, String)) && args(valor, nombre) ;

before(int valor, String nombre):
capturaCallParametros(valor,nombre) {
……
}
```

El advice before() puede acceder a los identificadores declarados en el pointcut capturaCallParametros(int, String) incluyendo valor y nombre en su signatura y luego, enlazando esos identificadores al pointcut capturaCallParametros.

- Ejecución de método: Captura la ejecución del cuerpo de un método.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : call (<modificador opcional> <tipo retorno> <clase>.<metodo>(<tipos parámetros>));

Ejemplo:

pointcut nombrePointcut ( ) : execution (* *.metodo(String)) ;

Con este pointcut se capturarían la ejecución de los métodos con el nombre "metodo" de cualquier clase, que devuelva cualquier valor, y que además tenga un parámetro de tipo String.

El pointcut execution tiene dos características:

- El contexto de el joinpoint es el del método de la clase que se captura.
- La signatura puede contener caracteres comodín para seleccionar un rango de joinpoints sobre diferentes clases y métodos.

Puede parecer que el pointcut de ejecución de método no aporta nada nuevo respecto al pointcut llamada a método. La diferencia entre ambos reside en dónde se invoca al advice correspondiente y el contexto. En el caso del pointcut de llamada a método, el advice es invocado donde se invoca el método, y el contexto es el de la clase que llama. Mientras que en el pointcut de ejecución de método, el advice es invocado una vez que se ha entrado al método, y por lo tanto, el contexto es el del método ejecutado.

Se puede utilizar en el advice el objeto apuntado por la referencia this de Java, creando un pointcut que especifique un parámetro del mismo tipo que la referencia this que se quiere capturar. Se debe utilizar conjuntamente los pointcuts execution(signatura) y this(Tipo | Identificador) para capturar la ejecución de un método y luego enlazar el identificador al objeto al que la referencia this apunta durante la ejecución del método.

Ejemplo:

```
pointcut capturaThisEnEjecucion (MiClase miObjeto) :
execution(void MiClase.log(int, String)) && this(miObjeto) ;

before(MiClase miObjeto) : capturaThisEnEjecucion (miObjeto)
{
    ……
}
```

El advice before puede acceder al identificador que hace referencia al objeto apuntado por this durante la ejecución del método incluyendo el identificador miObjeto en la signatura y luego enlazandolo al pointcut capturaThisEnEjecucion(MiClase).

- Llamada a constructor: Captura la creación de un nuevo objeto.

  Sintaxis:

  ```
  pointcut <nombrePointcut> (<valores a capturar>) : call
  (<modificador opcional> <clase>.new(<tipos parámetros>));
  ```

  Ejemplo:

  ```
  pointcut nombrePointcut ( ) : call (MiClase.new(String)) ;
  ```

Con este pointcut se capturarían las llamadas al constructor de la clase MiClase si el parámetro es de tipo String.

- Ejecución de constructor: Captura la ejecución de un nuevo objeto.

  Sintaxis:

  ```
  pointcut <nombrePointcut> (<valores a capturar>) : execution
  (<modificador opcional> <clase>.new(<tipos parámetros>));
  ```

  Ejemplo:

  ```
  pointcut nombrePointcut ( ) : execution (MiClase.new(String)) ;
  ```

  Con este pointcut se capturaría la ejecución del constructor de la clase MiClase si el parámetro es de tipo String.

- Inicialización de clase: Captura la inicialización de una clase.

  Sintaxis:

  ```
  pointcut <nombrePointcut> (<valores a capturar>) :
  staticinitialization (<clase>);
  ```

Ejemplo:

pointcut inicializacionClase( ) : staticinitialization (MiClase) ;

- Preinicialización de objeto: Captura cuando un objeto se va a inicializar usando un constructor que corresponde con una signatura específica.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>):preinitialization (<modificador opcional> <clase>.new(<tipos parámetros>));

Ejemplo:

pointcut preinicializacionClase( ) :
preinitialization (MiClase.new (int, String)) ;

Los joinpoints asociados al pointcut de tipo preinicialización de objeto ocurrirán después de capturar el constructor y antes de que sea llamada cualquier superclase.

Debido a una limitación en el compilador AspectJ el pointcut de preinicialización de objeto no puede utilizarse asociado con un around advice.

- Inicialización de objeto: Captura cuando se inicializa un objeto, invocado por un constructor con una signatura específica.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : initialization (<modificador opcional> <clase>.new(<tipos parámetros>));

Ejemplo:

pointcut inicializacionClase( ) :initialization (MiClase.new (int, String)) ;

Los joinpoint correspondientes a un pointcut de este tipo ocurrirán después de la inicialización de cualquier superclase y antes del return del método constructor.

Debido a una limitación en el compilador AspectJ el pointcut de preinicialización de objeto no puede utilizarse asociado con un around advice.

- Lectura de un atributo: Captura cuando un atributo de un objeto es accedido.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : get (<modificador opcional> <tipo> <clase>.<campo>));

Ejemplo:

pointcut accesoAtributo( ) : get (String MiClase.nombre) ;

Este pointcut tiene las siguientes características:

- Captura los accesos a un atributo, no solo las llamadas a los métodos de acceso a atributos.
- El pointcut lectura de atributo no puede capturar el acceso a atributos estáticos, aún cuando es perfectamente legal en AspectJ definir un pointcut de este tipo.
- Se pueden utilizar caracteres comodín para seleccionar un rango de diferentes atributos.
- Si queremos capturar el valor del atributo al que se está accediendo y utilizarlo en un advice, se utilizará el after() returning(<Valor de retorno>) advice con un identificador en la parte returning() que contendrá el valor que ha sido accedido.

Ejemplo:

```
pointcut capturarValorGet( ) : get (String
MiClase.nombre);
                        after( ) returning(String valor) :
capturarValorGet( ) {
            ……..
        System.out.println ("Valor campo: " + valor);
}
```

- Escritura de un atributo: Captura cuando se modifica un atributo de un objeto.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : set (<modificador opcional> <tipo> <clase>.<campo>));

Ejemplo:

pointcut accesoAtributo( ) : set (String MiClase.nombre) ;

Este pointcut tiene las siguientes características:

- No puede capturar la modificación de atributos estáticos, aún cuando es perfectamente legal en AspectJ definir un pointcut de este tipo.
- La signatura puede contener caracteres comodín para seleccionar un rango de diferentes atributos.
- Para capturar el valor de un campo después de haber sido modificado y así utilizarlo en un advice, se utiliza la combinación del pointcut args(Tipos | Identificadores) con el pointcut set(Signatura).

  Ejemplo:

```
pointcut capturarValorSet (String nuevoValor) :
                                set (String MiClase.nombre) &&
args(nuevoValor);

before (String nuevoValor) : capturarValorSet
(nuevoValor) {
        System.out.println ("Valor de campo
modificado a: " +                       nuevoValor);
}
```

- Ejecución de manejador de excepciones: Captura el manejador de una excepción particular.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : handler (<clase>)

Ejemplo:

pointcut excepcionPointcut ( ) : handler (IOException) ;

El pointcut de ejecución de manejador de excepciones tiene las siguientes características:

- El advice solo será aplicable si el patrón de tipo especifique Throwable o una subclase de Throwable.

- Solo el tipo de advice before( ) es soportado en este tipo de pointcut. No se podrá sobreescribir el comportamiento de un bloque catch utilizando el tipo de advice around( ).
- La signatura podrá contener caracteres comodín para seleccionar un mayor rango de joinpoints.

- Ejecución de advice: Captura la ejecución de todos los advice.

  Sintaxis:

  pointcut <nombrePointcut> : adviceexecution ( );

  Ejemplo:

  pointcut capturarEjecucionAdvice( ) : adviceexecution ( ) && !within(miAspecto);

  Al capturar absolutamente todos los advice debe ser utilizado junto a otros pointcut para aplicarlo donde se desee. En el ejemplo anterior se ha añadido el pointcut within negado para evitar la recursividad, debido a que el advice asociado a capturarEjecucionAdvice también sería capturado por el pointcut. Sin embargo, no se puede capturar la ejecución de un único advice.

  Si se quiere acceder al joinpoint original al que pertenece el advice correspondiente, habrá que añadir el identificador JoinPoint en la definición del pointcut. Además, se enlazará el identificador JoinPoint como parámetro al pointcut args(Tipos | Identificadores).

  Ejemplo:

```
pointcut capturarEjecucionAdvice (JoinPoint joinpointOriginal ) :
adviceexecution( ) && args(joinpointOriginal) && !within(miAspecto);

before(JoinPoint joinpointOriginal)
:adviceexecution(joinpointOriginal){
        ……
        }
```

- Joinpoints dentro de una clase o paquete: Captura todos los joinpoints que se encuentran en una clase o paquete específico.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : within (<clase>);

Ejemplo:

pointcut withinAdvice( ) : within (miClase);

Se pueden utilizar los caracteres comodín. Por ejemplo, within(miClase+) captura los joinpoints dentro de la clase miClase y sus subclases.

Un uso habitual de within es excluir los joinpoints generados en un aspecto para evitar la recursividad.

- Joinpoints dentro de método: Captura todos los joinpoints que se encuentran dentro de los métodos o constructores especificados.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : withincode (<modificador> <clase/constructor>.<método> (<tipos parámetros>));

Ejemplo:

pointcut withincodeAdvice( ) : withincode (* miClase.get*(String));

El ejemplo anterior captura todos los joinpoints dentro del cuerpo de los métodos de la clase miClase que empiezan por get.

Este pointcut no suele utilizarse por separado, sino conjuntamente con otros poincut para especificar mejor el rango de joinpoints afectados por un advice.

- Joinpoints dentro de una clase o paquete: Captura todos los joinpoints que se encuentran en una clase o paquete específico.

Sintaxis:

pointcut <nombrePointcut> (<valores a capturar>) : within (<clase>);

Ejemplo:

pointcut withinAdvice( ) : within (miClase);

Se pueden utilizar los caracteres comodín. Por ejemplo, within(miClase+) captura los joinpoints dentro de la clase miClase y sus subclases.

- Joinpoints dentro del flujo de control: Captura todos los joinpoints en el flujo de control de los joinpoints capturados por el pointcut pasado como parámetro.

  Sintaxis:
  pointcut <nombrePointcut> (<valores a capturar>) : cflow
  (<pointcut>);

  Ejemplo:
  pointcut inicialPointcut( ) : call (void MiClase.getNombre(String));
  pointcut cflowPointcut( ) : cflow (inicialPointcut( ));

- Joinpoints dentro del flujo de control, excluyendo el joinpoint inicial: Captura todos los joinpoints en el flujo de control de los joinpoints capturados por el pointcut pasado como parámetro, sin incluir el joinpoint inicial del flujo de control.

  Sintaxis:
  pointcut <nombrePointcut> (<valores a capturar>) :
  cflowbelow (<pointcut>);

  Ejemplo:
  pointcut inicialPointcut( ) : call (void MiClase.getNombre(String));

  pointcut cflowbelowPointcut( ) : cflowbelow (inicialPointcut);

- Instancia de un objeto: Captura todos los joinpoints cuyo objeto actual (el objeto ligado a this) es una instancia del tipo especificado, o es una instancia de una clase que coincide con la clase asociada al identificador.

  Sintaxis:
  pointcut <nombrePointcut> (<valores a capturar>) :
  this (<tipo> o <identificador> o *);

  Ejemplo:
  pointcut thisPointcut(MiClase objeto) : this (objeto);

  El pointcut this tiene las siguientes características:

  - El tipo indicado como parámetro debe corresponder a una clase válida puesto que no se permite el uso de caracteres comodín.

- El identificador se utiliza para examinar el tipo del objeto referenciado y exponer dicho objeto al advice correspondiente.
- Usando el comodín * como parámetro permite que el objeto apuntado por this sea un objeto válido, pero sin importar el tipo.
- El descriptor this no captura llamadas a métodos estáticos porque carecen de un objeto actual. Para capturar los joinpoints asociados a métodos estáticos se puede utilizar el pointcut within con el operador '+', y así capturar los métodos estáticos y los de instancia de la clase en cuestión y de sus subclases.

- Instancia de un objeto destino: Captura todos los joinpoints cuyo objeto destino (el objeto sobre el cual se invoca un método o una operación de atributo) es una instancia del tipo especificado, o es una instancia de una clase que coincide con la clase asociada al identificador.

Sintaxis:
pointcut <nombrePointcut> (<valores a capturar>) : target (<tipo> o <identificador> o *);

Ejemplo:
pointcut targetPointcut(MiClase objeto) : target (objeto);

El pointcut target tiene las siguientes características:

- El tipo indicado como parámetro debe corresponder a una clase válida puesto que no se permite el uso de caracteres comodín.
- El identificador se utiliza para examinar el tipo del objeto referenciado y exponer dicho objeto al advice correspondiente.
- Usando el comodín * como parámetro permite que el objeto apuntado por target sea un objeto válido, pero sin importar el tipo.
- El descriptor target no captura llamadas a métodos estáticos ni manejadores de excepciones porque

carecen de un objeto destino. Para capturar los joinpoints asociados a métodos estáticos se puede utilizar el pointcut within con el operador '+', y así capturar los métodos estáticos y los de instancia de la clase en cuestión y de sus subclases.

- Argumentos: Captura todos los joinpoints si los argumentos del joinpoint son de un determinado tipo, número u orden.

Sintaxis:
pointcut <nombrePointcut> (<valores a capturar>) : args (<tipos> o <identificadores>, .., <repeat>);

Ejemplo:
pointcut argsPointcut(MiClase objeto) : args (objeto);

El pointcut args tiene las siguientes características:

- El identificador se utiliza para examinar el tipo de los objetos en tiempo de ejecución que son los argumentos del joinpoint capturado y exponer esos objetos al advice correspondiente.
- Usando .. permite mayor flexibilidad en el número de argumentos indicado. Si no se utiliza, el pointcut captura los joinpoints que exactamente corresponden con los tipos especificados, es decir, solo aquellos joinpoints cuyos argumentos coinciden en orden, tipo y número. Solo puede utilizarse un único .. en un pointcut args. Ejemplo:
  args(MiClase,..,float)
  capturará aquellos joinpoints que tengan dos argumentos, el primero debe ser un objeto de tipo MiClase, seguido de cualquier número de argumentos, y entre ellos debe haber un float.
- El comodín * permite flexibilidad para indicar el tipo de argumento pero no el número de argumentos, el cual debe coincidir con el indicado. Así, args(*) capturará los joinpoints que contengan un único argumento de cualquier tipo, y args(*,*) capturará los joinpoints que contengan dos argumentos de cualquier tipo.

- Los pointcuts get, set y staticinitialization no tienen argumentos para exponer utilizando el pointcut args.
- Si se utiliza Object como tipo, el tipo primitivo correspondiente al objeto será pasado al advice.

- Condición en tiempo de ejecución: captura un advice si es true el resultado de un valor en tiempo de ejecución en el momento de ser capturado un joinpoint.

Sintaxis:
pointcut <nombrePointcut> (<valores a capturar>) : if (<expresión booleana>);

Ejemplo:
```
pointcut  ifPointcut(MiClase objeto) : if ( thisJoinPoint.getThis( )
instanceof  MiClase &&
 ((MiClase) thisJoinPoint.getthie( )).getSalario( ) < salarioMinimo)&&
!withincode(* MiClase.get*( ));
```

Advice

Los pointcuts definen los joinpoints en los que se tiene interés, mientras que los advice definen qué hacer al llegar a esos joinpoints.

Para acceder a los atributos y métodos de un objeto particular desde dentro de un advice hay que pasar el objeto correspondiente al advice como un parámetro en la declaración del pointcut.

Ejemplo:

```
public privileged aspect accesoAdvice {

        pointcut ejecucionPointcut( ): execution (void
MiClase.getNombre(String));

        after (MiClase miClase) : ejecucionPointcut ( ) && this
(miClase) {

            miClase.setX(2.0f);

            System.out.println(miClase.x);

        }
```

```
}
```

Un objeto de la clase MiClase es accesible desde el advice usando la definición de pointut this(Identificador). El método setX es llamado desde dentro del advice y muestra el acceso a métodos públicos desde el advice. Para tener acceso al atributo privado x es necesario un cambio en la estructura del aspect ya que se está intentantdo romper la encapsulación. Este cambio consiste en declarar el aspecto como privileged, lo que permite un acceso a la clase completo y sin restricciones, incluyendo variables y métodos no declarados como públicos. Este tipo de acceso debe ser utilizado con cuidado porque puede producir problemas indeseados.

Para acceder al contexto del joinpoint desde dentro del advice se utilizan las variables thisJoinPoint y thisJoinPointStaticPart.

Las clases en Java tienen una variable this que permiten referenciar a los objetos y trabajar con ellos. El compilador de AspectJ convierto los aspects en clases, por lo tanto, los aspects también tiene una referencia this. Sin embargo, un contexto de joinpoint adicional puede ser accedido desde el punto de vista del joinpoint capturado. AspectJ dispone de la variable thisJoinPoint para exponer este contexto de joinpoint. Además, dispone de la variable thisJoinPointStaticPart para acceder unicamente a la parte estática del contexto de un joinpoint. Si queremos acceder a la información estática de un joinpoint se recomienda usar la variable thisJoinPointStaticPart por motivos de rendimiento.

thisJoinPoint dispone de los siguientes métodos para acceder a la información estática y dinámica del contexto de un joinpoint a través de los siguientes métodos:

- getSignature ( ): devuelve un objeto de la clase Signature que representa la signatura del joinpoint. A partir de este objeto se puede acceder al identificador del joinpoint (getName ( )), a sus modificadores de acceso (getModifiers ( )), etc.

- getSourceLocation ( ): devuelve un objeto de la clase SourceLocation que representa al objeto invocador del joinpoint. A partir de este objeto se puede acceder a la clase del objeto invodador con el método getWithinType ( ).

- toString ( ): devuelve una cadena que representa el joinpoint.

- toShortString ( ): devuelve una cadena corta que representa el joinpoint.
- toLongString ( ): devuelve una cadena larga que representa el joinpoint.
- getStaticPart ( ): devuelve el objeto que encapsula la parte estática del joinpoint.
- getTarget ( ): devuelve el objeto destino.
- getThis ( ): devuelve el objeto actual.
- getArgs ( ): devuelve los argumentos del joinpoint.
- getKind ( ): devuelve una cadena que representa la categoría del joinpoint (por ejemplo, method-call).

AspectJ soporta tres tipos de advice que indican cuándo se ejecutará el advice correspondiente al joinpoint capturado. Estos tres tipos son: before, after y around.

Advice before:

El cuerpo del advice se ejecuta antes del joinpoint capturado. Si se produjera una excepción en el cuerpo del advice, el joinpoint capturado no se llegaría a ejecutar.

Ejemplo:

```
pointcut llamadaPointcut ( ): call (void
MiClase.getNombre(String));

before ( ) : llamadaPointcut ( ) {

…..

}
```

Advice after:

El cuerpo del advice se ejecuta después del joinpoint capturado. Si se produjera una excepción en el cuerpo del advice el joinpoint capturado no se llegaría a ejecutar.

Ejemplo:

```
pointcut llamadaPointcut ( ): call (void
MiClase.getNombre(String));

after ( ) : llamadaPointcut ( ) {

   …..

}
```

Existen tres tipos de advice after:

- After no calificado: se ejecuta siempre, sin importar si el joinpoint asociado terminó correctamente o con el lanzamiento de una excepción. Equivaldría al bloque finally de Java.

- After returning: solo se ejecutará si el joinpoint asociado termina correctamente, pudiendo acceder al valor de retorno devuelto por el joinpoint. El advice no se ejecutará si se produce una excepción. El advice after returning solo se ejecutará si el tipo del valor de retorno es compatible por el definido en la cláusula returning. Si se indica el tipo Object en la cláusula returning, el advice se ejecutará para cualquier tipo devuelto.
El valor devuelto por los joinpoints de llamada o ejecución de método será el del método correspondiente, si el joinpoint es llamada a constructor el valor será el objeto creado, y para el joinpoint de lectura de atributo el valor será el atributo implicado. Para el resto de joinpoints el valor devuelto será null. Los joinpoints de manejador de excepciones no pueden manejarse hasta el momento por un advice after returning.

- After throwing: el advice solo se ejecutará si el joinpoint termina con el lanzamiento de una excepción. Al igual que en after returning el advice solo se ejecutará si el tipo de la excepción lanzada por el joinpoint coincide con el indicado en la cláusula throwing.

Advice around:

El cuerpo del advice se ejecuta en lugar del joinpoint capturado, reemplazando la ejecución original. Con este tipo de advice también se puede ignorar la ejecución del joinpoint si los parámetros no cumplen ciertas condiciones, o cambiar el contexto del joinpoint (argumentos, el objeto actual this o el objeto destino target).

Ejemplo 1:

```
pointcut llamadaPointcut1 ( ): call (int MiClase.getX( ));
int around (int valor) : llamadaPointcut1 (valor) {
    …..
    return proceed (valor)
}
```

La llamada proceed indica que se debería seguir ejecutando la lógica original del joinpoint. Se le puede pasar un parámetro (como en el ejemplo 1), y éste debe ser pasado en el proceed.

Ejemplo 2:

```
pointcut llamadaPointcut2(int valor ):call (void MiClase.setX( ));
void around (int valor) : llamadaPointcut2 (valor) && args(valor) {
    if (valor >= MIN_VALOR) {
        proceed (valor);
    }
}
```

Como ya comentamos, también podemos utilizar el advice around para no ejecutar el joinpoint original si el parámetro pasado valor no es mayor que la constante MIN_VALOR como en el Ejemplo 2.

Ejemplo 3:

```
pointcut llamadaPointcut3 (Object mens): call (public void print*(*));
void around (Object mens) : llamadaPointcut3 (valor) && args(mens) {
    proceed (“Version nueva “ + mens);
}
```

En este ejemplo 3 se muestra cómo utilizar el advice around para modificar el contexto del joinpoint capturado, en concreto el parámetro

pasado. Este advice añade el literal "Versión nueva" a todos los mensajes que se imprimen por pantalla.

Ejemplo 4:

```
pointcut operacionesExcepcionPointcut ( ) : handler (IOException) ;
Object around ( ) : operacionesExcepcionPointcut ( ) {
   try {
        return proceed ( );
   }catch (Exception e) {
   // Manejo de la excepción
   }
}
```

El ejemplo 4 muestra otra aplicación del advice around para manejar un grupo de excepciones de la misma manera sin tener que repetir el código en los correspondientes try/catch.

3 DESARROLLO TEÓRICO

3.1 CONFLICTOS

Un sistema orientado a aspects debe cumplir una serie de propiedades [2]:

- Cada aspect deber ser claramente identificable, auto-contenido y fácil de cambiar.
- Los aspects no deben interferir entre ellos ni con la funcionalidad del sistema.

Uno de los problemas que se plantean en el desarrollo de software orientado a aspects es la interferencia, interacción o conflicto entre aspects (de ahora en adelante hablaremos de conflicto entre aspects). Se dice que existe conflicto entre aspects cuando un joinpoint está asociado a diferentes pointcuts pertenecientes a distintos aspects, esto es, que diferentes aspects piden activarse en el mismo momento.

Esta situación puede no ser un problema si los aspects son independientes, es decir, que el comportamiento representado en cada aspect es independiente del resto, por lo que el comportamiento del sistema no se ve afectado.

El siguiente ejemplo muestra dos aspects independientes. Ambos se activarán al ejecutarse el método metodo1 de la clase clase1. Sin embargo, no existe conflicto porque un aspect escribirá en un archivo, mientras que el otro aspect escribirá en otro archivo distinto. Sea cual sea el orden de activación de los aspect el resultado será el mismo.

```
aspect Aspect1 {
        void before ( ): call (public void clase1.* ( )) {
           …..
           // Escritura en archivo1
        }
}
```

```
aspect Aspect2 {
        void before ( ): call (public void clase1.metodo1( )) {
            …..
            //Escritura en archivo2
        }
}
```

Sin embargo, si los aspects no son independientes porque el comportamiento de alguno puede interactuar con el de otros y producir resultados no deseados o inconsistentes, entonces, decimos que existe conflicto entre aspects que habrá que resolver.

En el siguiente ejemplo se muestran dos aspects entre los que hay un conflicto.

```
aspect Aspect1 {
        void around ( ): call (public void clase1.metodo1( )) {
            …..
            proceed;
        }
}
aspect Aspect2 {
        void around ( ): call (public void clase1.metodo1( )) {
            …..
            return;
        }
}
```

Considerando que en los requerimientos del sistema los advice de ambos aspect deben ejecutarse, si no hay declarada una precedencia que indique qué aspect se debe ejecutar primero, los advice podrían ordenarse de dos formas distintas. Podría ejecutarse primero el advice de Aspect1 y luego el advice de Aspect2, o podría ejecutarse primero el

advice de Aspect2 y luego el de Aspect1. El primer caso sería correcto, sin embargo si se ejecuta primero el advice del Aspect2, el advice del Aspect1 no llegaría a ejecutarse debido a la sentencia return.

Hay varias razones por las que se pueden producir conflictos [9]. Primero, el uso de patrones o comodines para englobar grupos de pointcuts (ver apartado 2.3.3.1), los cuales son muy útiles pero no se conoce el alcance que puede tener en el momento de crearlo. Nuevas funcionalidades añadidas a la aplicación podrían encajar con la especificación del pointcut, sin ser esa la intención.

Otra razón para producirse conflicto es el uso de diferentes dominios. Por ejemplo, si estamos realizando manejo de errores basándonos en un valor entero como retorno (0 resultado correcto, distinto de cero resultado incorrecto), podría ocurrir que un determinado pointcut cubriera métodos que realizan operaciones matemáticas, y por lo tanto, considerar como error operaciones que devuelvan un valor distinto de cero.

Por último, el uso de información dinámica en un pointcut también puede limitar el control sobre el resultado del pointcut. Por ejemplo, AspectJ tiene sentencias if para preguntar por datos en tiempo de ejecución. Esto hace que perdamos cierto control en tiempo de compilación.

Cuando surgen conflictos entre aspects es necesario determinar la prioridad de los aspects y políticas de activación para asegurar un correcto funcionamiento de la aplicación.

En la mayoría de los lenguajes orientados a aspectos la detección, control y resolución de conflictos es totalmente manual. En general, el wover o tejedor de aspectos, ante un conflicto entre aspects procederá sin darse cuenta de tal conflicto y, por lo tanto sin dar ningún aviso de conflicto.

Cuando el sistema es pequeño y tiene pocos aspects, la detección y resolución de conflictos es en cierto modo manejable. El problema viene cuando el sistema es grande, hay muchos aspects y el desarrollo lo han desarrollado diferentes personas y en diferentes momentos, o cuando hay que modificar o eliminar funcionalidad o aspects. El control de conflictos se puede volver una tarea difícil de llevar y puede incluso

producir resultados desastrosos en el sistema. Sin un control de conflictos el sistema se vuelve impredecible y difícil de manejar porque en cualquier momento un simple cambio puede hacer que el sistema se comporte de manera incorrecta. Por ejemplo, distintas ejecuciones pueden producir distintos resultados o distintas versiones del wover o tejedor podrían comportarse de diferente forma ante un conflicto.

Con todo esto, resulta necesario poder administrar los conflictos, detectando posibles conflictos e incluso, si es posible, resolverlos automáticamente. Esta administración de conflictos puede entenderse, por ejemplo, como un paso anterior al wover que pre-procese el código e informe al desarrollador de los posibles conflictos. Con esto, se libera al programador de la difícil tarea de controlar los conflictos, garantizando mayor seguridad en la aplicación.

3.2 Detección y Resolución de conflictos

3.2.1 Aplicación libre de conflictos

Teniendo en cuenta las interacciones producidas entre un advice y los métodos de una clase, Rinard [21] clasifica los advices en:

- Augmentation: si el método asociado siempre se ejecuta. Por ejemplo en aspects de logging y monitorización.
- Narrowing: si el advice decide si el método asociado se ejecuta o no, por ejemplo a través de precondiciones que comprueba el advice antes de permitir o no la ejecución del método.
- Replacement: si el advice reemplaza al método asociado.
- Combinación: si el método y el advice interactúan de cualquier otra forma.

Rinard también define scopes como conjunto de campos que son accedidos por un objeto o por un aspect, y presenta los siguientes tipos de interacciones:

- Observación: si un aspect lee datos modificados por otro aspect.
- Actuación: si un aspect modifica datos que son leídos por otro aspect

- Combinación: si dos aspect modifican datos compartidos.

Además, si dos aspects leen los mismos datos, los aspects serán independientes, mientras que si no comparten joinpoints, los aspects son ortogonales.

Así, según esta clasificación una aplicación libre de conflictos será aquella con aspects ortogonales o independientes, con interacciones del tipo Observación, y donde los advices sean del tipo Augmentation o Narrowing.

3.2.2 Clasificación de Conflictos

Clasificación según elementos involucrados

Tessier [20] clasifica los conflictos siguiendo varios criterios. Dependiendo de los elementos involucrados:

- Conflictos intra-aspects: si el conflicto se produce entre advices dentro del mismo aspect.

- Conflictos inter-aspects: si el conflicto es entre dos advices de diferentes aspects

- Conflictos aspect-clase: si el conflicto se produce entre un aspect y una clase.

Clasificación según la Composición

Existen diferentes estrategias que implican distintos joinpoints para relacionar aspects y otros componentes dentro del sistema (ver Figura 4). Así, hablamos de:

- Composición de clase: el código del aspect se activa cuando todos los objetos de una determinada clase reciben un mensaje.

- Composición de objeto: el código del aspect se activa cuando un determinado objeto recibe un mensaje.

- Composición de método: el código del aspect se activa cuando todos los objetos de una clase reciben un mensaje de un método en concreto.

- Composición de objeto-método: se trata de una mezcla entre la composición de objeto y la de método, en la cual el código del aspect se activa cuando un determinado objeto recibe un mensaje de un método en concreto.

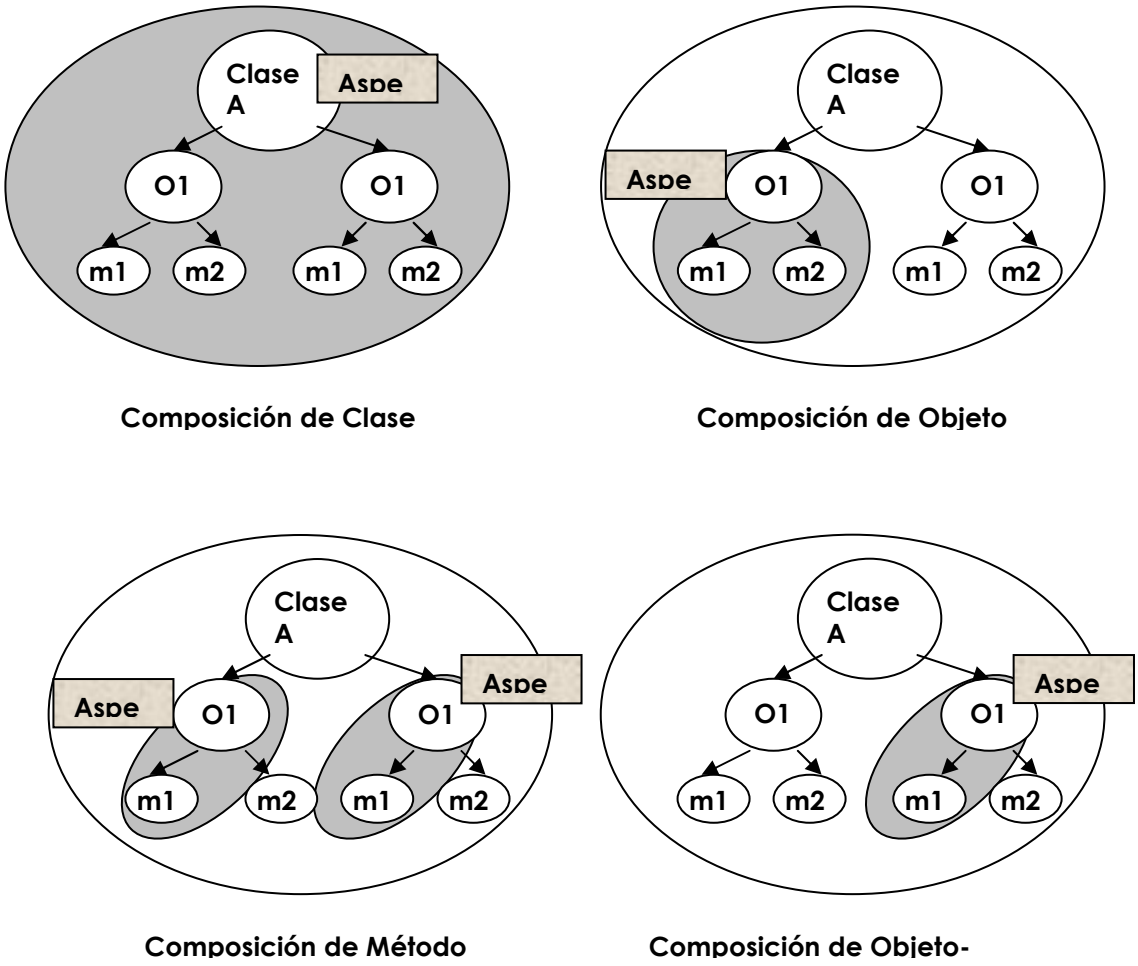


Figura 4. Tipos de Composición

Partiendo de esta clasificación podemos hacer la siguiente clasificación de conflictos (ver Figura 5):

- Conflicto clase-objeto: si el dominio de activación de un aspect asociado a un objeto está incluido en el dominio de activación de otro aspect asociado a la clase a la que pertenece el objeto del primer aspect.

- Conflicto clase-método: si el dominio de activación de un aspect asociado a un método se encuentra incluido en el dominio de

activación asociado a la clase del objeto que invoca el método. Solo se producirá conflicto cuando se invoque a ese método en concreto.

- Conflicto clase- método/objeto: si el dominio de activación de un aspect asociado al método de un objeto en concreto está incluido en el dominio de activación de la clase a la que pertenece el objeto. Este tipo tiene menos peligro que el anterior puesto que solo se producirá conflicto cuando se invoque al método asociado de un objeto determinado.

- Conflicto método-método: si dos aspects están asociados al mismo método.

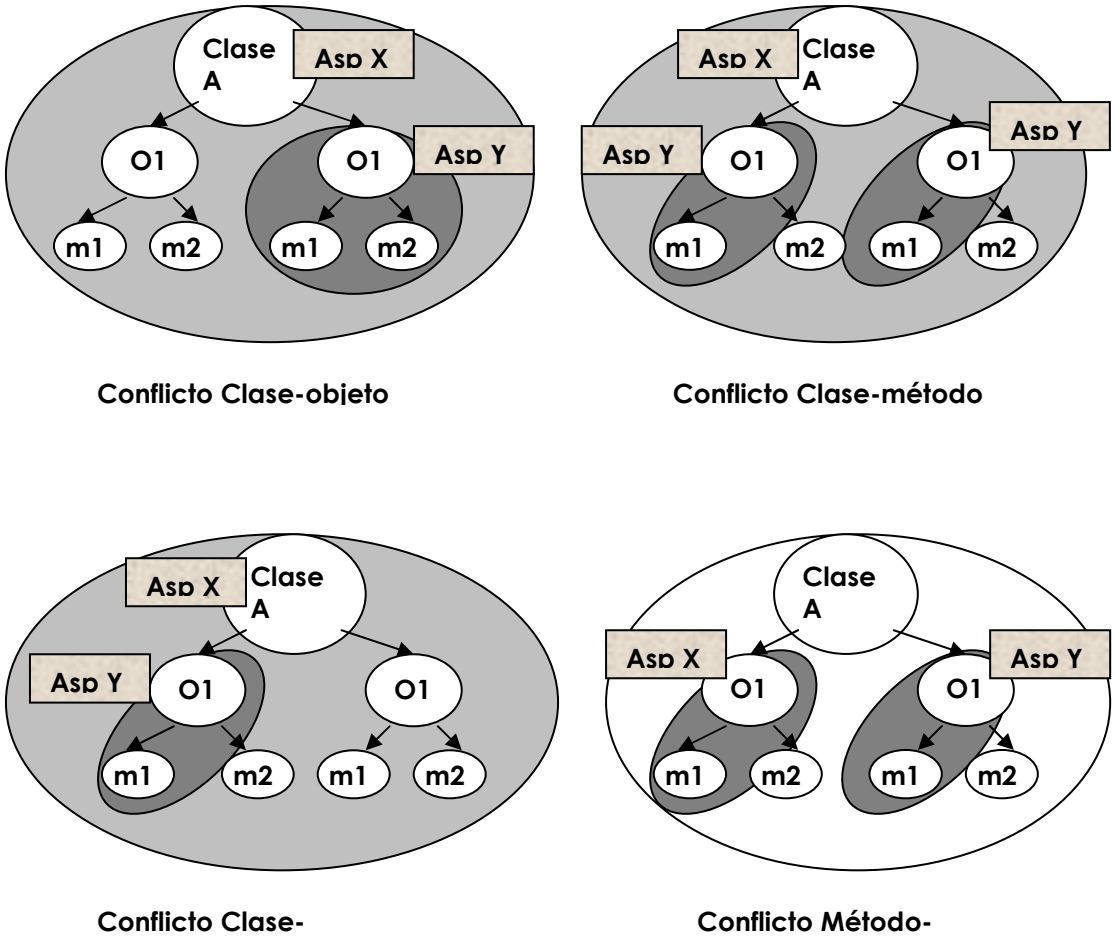


Figura 5. Tipos de Conflictos según Composición

Clasificación según Tipo

Dependiendo de en qué momento se detecten los conflictos, éstos pueden clasificarse como:

- Conflicto estático: si se detecta al establecerse la relación en tiempo de compilación.
- Conflicto dinámico: si se detecta el conflicto en tiempo de ejecución, por ejemplo durante la fase de pruebas de la aplicación.

Conflictos potenciales

El desarrollador debe poder definir los conflictos potenciales y la forma de solucionarlos de acuerdo al tipo de conflicto y el dominio de aplicación, pudiendo determinar las prioridades y las políticas de activación [9].

A veces la lista de conflictos potenciales puede ser muy extensa y puede que solo un subconjunto de esta lista requiera una resolución diferente o que haya que analizar con más detalle alguno de los conflictos.

Dados dos aspects, asp1 y asp2, los siguientes casos podrían ser conflictos:

- asp1 y asp2 están asociados a la misma clase
- asp1 y asp2 están asociados al mismo método y a la misma clase
- asp1 y asp2 están asociados al mismo objeto
- asp1 y asp2 están asociados al mismo objeto-método
- asp1 está asociado a una clase y asp2 está asociado a un método de esa misma clase
- asp1 está asociado a una clase y asp2 está asociado a un objeto de esa misma clase
- asp1 está asociado a una clase y asp2 está asociado a un objeto-método de una instancia de esa misma clase
- asp1 está asociado a un método y asp2 está asociado a una instancia de la clase a la que pertenece el método

- asp1 está asociado a un método y asp2 está asociado a un objeto-método correspondiente al mismo método e instancia de la clase
- asp1 está asociado a un objeto y asp2 está asociado a un objeto-método del mismo objeto.

Taxonomía de Conflictos

Cuando se detecta un conflicto la acción a realizar dependerá de la aplicación. Así, a veces bastará con dar prioridad a los aspects para que se ejecuten en un determinado orden, y otras veces habrá que desactivar uno o varios aspects.

Se han identificado seis categorías [2] para la activación de políticas cuando se detecta un conflicto que permiten resolver una gran cantidad de situaciones con conflictos. Estas categorías son:

- De Orden: se establece una declaración de precedencia para los aspects involucrados.
- De Orden Inverso: se invierte la declaración de precedencia establecida para los aspects involucrados.
- Opcional: el sistema decide qué aspects ejecutar dependiendo de alguna política o de forma aleatoria.
- De Exclusión: se ejecuta solo uno de los aspects en caso de que los aspects involucrados sean contradictorios.
- De Nulidad: se anula la ejecución de ambos aspects.
- Dependiente del Contexto: se decidirá la ejecución de los aspects dependiendo del contexto en el que se encuentren en tiempo de ejecución.

Estas seis categorías se podrían resumir en cuatro grandes grupos que identifican la política de activación de los aspects:

- Nulo: ningún aspect será activado
- Exclusivo: solo un aspect será activado
- Parcial: solo unos determinados aspects serán activados
- Total: todos los aspects serán activados

Orden de activación

Si varios o todos los aspects que intervienen en el conflicto se van a activar, habrá que decidir el orden en que se activarán los aspects para que el comportamiento del sistema sea el correcto:

- Prioridad: cada aspect tiene una prioridad con respecto al resto de aspects
- Acceso a datos: se puede elegir, como en bases de datos, que tengan mayor prioridad los accesos de lectura, luego los de lectura/escritura, y por último los de escritura.
- Especificidad: se trata de dar mayor prioridad a los aspects con dominio más específico. Esto es, primero se ejecutaría los aspects método-objeto.
- Precedencia: se define un orden de precedencia entre los aspects.

3.2.3 Detección y Resolución de conflictos con Unidades de Prueba

En programación orientada a objetos los conflictos se previenen con técnicas de encapsulación como Pruebas Unitarias (Unit Testing). La encapsulación permite hacer saber a los desarrolladores que aunque se cambie el funcionamiento de una unidad, esto no estropeará otras partes del sistema, siempre y cuando no se modifiquen las interfaces públicas.

La mayoría de los lenguajes AOP permiten cambiar el comportamiento interno de la unidades, lo cual es una importante característica de la AOP. Sin embargo, la encapsulación deja de ser una garantía para asegurar que el sistema está libre de conflictos.

Además, las pruebas unitarias se vuelven dificiles de utilizar en AOP debido a que los aspects pueden cambiar el comportamiento público de las unidades, y por lo tanto, las pruebas unitarias pueden pasar a ser obsoletas al no adecuarse al nuevo comportamiento. De aquí que, las pruebas unitarias deban adaptarse a la nueva forma de programación.

André Restivo y Ademar Aguiar proponen una metodología [25] basada en el uso de pruebas unitarias para mejorar el tratamiento de conflictos en AOP. Con ella se pretende adaptar las pruebas unitarias a

la programación orientada a aspects, de forma que se pueda seguir utilizando como metodología de pruebas de regresión, a la vez que ayude a detectar los conflictos entre aspects.

En AOP debería poder especificarse las pruebas unitarias válidas para que un aspect se considere correcto al componerse con el sistema. Igualmente, debería ser posible determinar qué pruebas unitarias va a romper un aspect.

En ocasiones, los aspects dependen unos de otros. Esto sucede cuando un aspect necesita cierto comportamiento que se encuentra presente en otro aspect. Debería ser posible introducir nuevas pruebas unitarias indicando el comportamiento de los aspects.

En la figura 6 se puede ver un ejemplo de un diagrama donde aspects añaden, eliminan y dependen de pruebas unitarias. Los cuadrados grises representan pruebas unitarias y los círculos representan los aspect. Las flechas identifican qué unidad o aspect creó la prueba. Las líneas con un círculo identifican una relación de dependencia. Y las líneas con el rombo identifican qué prueba invalida un aspect.

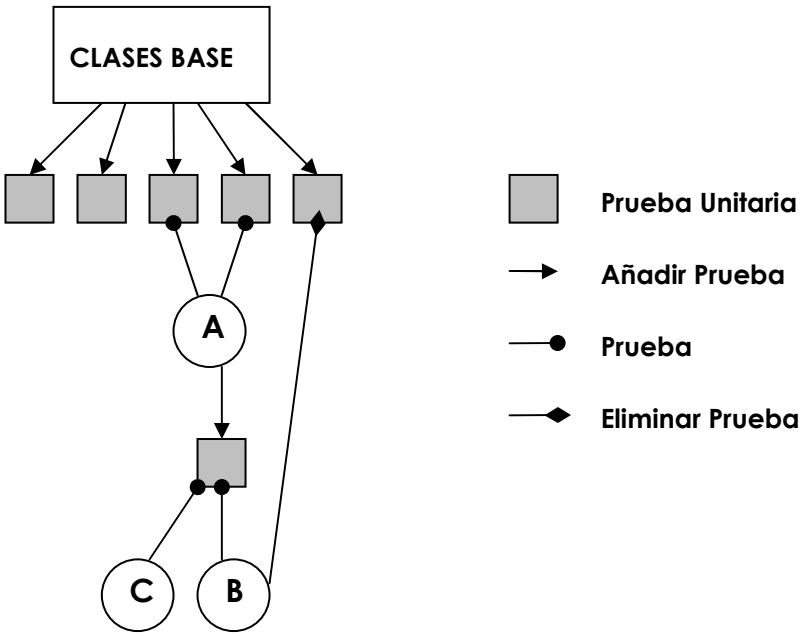


Figura 6. Aspects y Pruebas Unitarias

Observando la figura 6 se pueden extraer la siguiente información. Inicialmente el sistema se componía de cinco pruebas unitarias. El

aspect A depende de dos de esas pruebas unitarias, y además, crea una nueva prueba unitaria. El aspect B depende de la prueba unitaria creada por el aspect A, además de invalidar una prueba unitaria. Por último, el aspect C depende también de la prueba unitaria creada por el aspect A.

También se pueden extraer conclusiones, como que los aspect A y B son aspect que añaden comportamiento al sistema. Que el aspect B probablemente modifica el comportamiento del código original y que los aspect B y C dependen del aspect A. También se puede concluir que los aspect deberían ejecutarse en el orden A, seguido de B, seguido de C.

Si las pruebas unitarias se utilizan correctamente, además de asegurarnos un correcto funcionamiento del sistema, pueden ayudarnos a detectar la mayoría de los conflictos introducidos por los aspects. La figura 7 nos muestra un ejemplo de conflicto entre los aspectos C y D, donde el aspect C está cambiando cierta funcionalidad que necesita el aspect D.

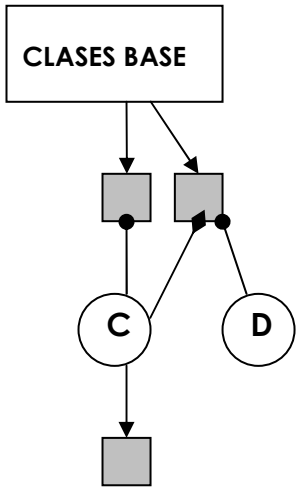


Figura 7. Conflicto usando Pruebas Unitarias

Para poder utilizar esta metodología el lenguaje de aspectos debería permitir introducir anotaciones con las que indicar qué aspects van a invalidar, de qué aspects dependen o qué aspects van a añadir funcionalidad al sistema.

3.3 AspectJ y los Conflictos

3.3.1 Conflictos según niveles de "Semejanza o Igualdad"

Los conflictos se pueden organizar según niveles de "semejanza o igualdad" [2] basándose en la estructura de los pointcut, distinguiendo entre conflictos de semejanza total o conflictos de semejanza parcial.

Para entender esta clasificación hay que aclarar el concepto de tipo de corte y tipo de advice:

- Tipo de corte en AspectJ son los cortes primitivos call, execution, get, set, inicialization, etc.
- Tipo de advice: before, after, around

Conflictos de semejanza total

Se dice que existe conflicto de semejanza total entre aspects si los cortes de los pointcut definidos coinciden en tipo de corte y joinpoints. Además, los joinpoints están asociados al mismo tipo de advice.

Ejemplo:

```
aspect A
{
        pointcut p1 ( ) : call (void clase.metodo( ));
        before ():  p1 ()
        {
           ……
        }
}
aspect B
{
        pointcut p2 ( ) : call (void clase.metodo( ));
        before ():  p2 ()
        {
```

```
        ……
    }
}
```

Conflictos de semejanza parcial

Existe conflicto de semejanza parcial entre aspects si los joinpoints coinciden y además están asociados al mismo tipo de advice.

Ejemplo:

```
aspect A
{
    pointcut p1 ( ) : call (void clase.metodo( ));
    before ():  p1 ()
    {
        ……
    }
}
aspect B
{
    pointcut p2 ( ) : execution (void clase.metodo( ));
    before ():  p2 ()
    {
        ……
    }
}
```

En este tipo de conflicto se puede producir un resultado impredecible puesto que al estar los joinpoints asociados a distintos tipos de corte, los aspects se activan en momentos de la ejecución diferentes

pero sobre el mismo contexto (objeto this, objeto target o argumentos de los métodos). Esto puede producir interferencia entre aspects.

La siguiente tabla (Tabla 3) muestra posibles tipos de corte que pueden producir conflicto:

Aspecto A	Aspecto B
p1 () : call (join point)	p2 () : execution (join point)
p1 () : set (join point)	p2 () : get (join point)
p1 () : call (join point)	p2 () : initialization (join point)
p1 () : call (join point)	p2 () : preinitialization (join point)
p1 () : execution (join point)	p2 () : initialization (join point)
p1 () : execution (join point)	p2 () : preinitialization (join point)

Tabla 3. Tipos de corte que producen conflictos potenciales

3.3.2 Detección y Resolución de Conflictos en AspectJ

AspectJ no tiene ningún mecanismo automático para detectar y resolver conflictos entre aspects, es decir, actúa como si no existieran. Es responsabilidad del desarrollador su detección y resolución.

AspectJ cubre las categorías de orden y opcional descritas en el apartado de Taxonomía de Conflictos. Por defecto no tiene ningún orden establecido, asi que habrá que indicarlo explicitamente mediante la claúsula "declare precedence", en otro caso, se ejecutarán los advices dependiendo de distintos factores, como por ejemplo, la versión del compilador de AspectJ.

AspectJ sigue una serie de reglas para determinar los advices que se ejecutarán:

- El aspect con mayor precedencia ejecutará su advice before sobre un joinpoint antes que otro de menor precedencia.

- El aspect con mayor precedencia ejecutará su advice after sobre un joinpoint después que otro de menor precedencia.

- El advice around con mayor precedencia engloba al de menor precedencia. De esta forma, el advice con mayor precedencia decide si el advice de menor precedencia se ejecutará según se utilize la llamada proceed(). Si el aspecto de mayor precedencia no hace la llamada a proceed() en el cuerpo del advice, el aspect de menor precedencia no se ejecutará, ni tampoco el joinpoint.

Para poder cambiar la precedencia de los advice, AspectJ tiene la cláusula "declare precedence" que al incluirse dentro de un aspect permite controlar el orden de los aspect.

La sintaxis de esta claúsula es:

declare precedence: typePattern1, typePattern2,… ;

cuyo resultado es que los advices de los aspects que cumplan el patrón typePattern1 tienen mayor precedencia que los advices de los aspects que cumplan el typePattern2, y así sucesivamente.

Esta claúsula de precedencia puede incluirse en alguno de los aspects involucrados, o en un aspect aparte que controle la precedencia entre los distintos aspects. La segunda opción es mejor ya que los aspects no tienen por qué conocer el resto de los aspect y permite una separación del control de precedencia que ayuda a su mantenimiento.

En la lista de typePatterns se pueden usar wildcards para referenciar a los aspects. Así, podemos escribir:

declare precedence: A*, B*

para indicar que los aspects que empiezan por A tienen mayor precedencia que los aspects que empiezan por B. Habrá que tener presente que con esta claúsula no se especifica ninguna precedencia entre los aspects que empiezan por A.

También se puede indicar que un aspect domina al resto, o viceversa.Así,

declare precedence: A, *

indica que el aspect A tiene precedencia sobre el resto de los aspect en el sistema. Mientras que:

declare precedence: *, A

indica que el aspect A es el de menor precedencia en el sistema.

También hay que tener cuidado de no insertar precedencias circulares que podrían producir errores de compilación. La siguiente claúsula produciría error de compilación:

declare precedence: A*, B, Ab

Sin embargo, se pueden especificar precedencias circulares sin que se genere error de compilación escribiendo dos claúsulas distintas:

declare precedence: A*, B;

declare precedence: B, Ab

Por último, hay que señalar que al igual que los aspects abstractos deben ser extendidos con aspects concretos, los advices se aplican sobre aspects concretos (no abstractos), por lo tanto, la sentencia "declare precedence" solo tendrá efecto sobre aspects concretos.

Así, si tenemos los aspects abstractos aspectAbstract1 y aspectAbstract2, y los aspects concretos derivados de éstos miAspect1 y miAspect2, respectivamente, serán válidas las siguientes sentencias:

Declare precedence: miAspect1, miAspect2;

Declare precedence: aspectAbstract1+, aspectAbstract2+;

La última sentencia indica que los aspects derivados de aspectAbstract1 tienen prioridad sobre los aspects derivados de aspectAbstract2.

Herencia y precedencia

AspectJ especifica implicitamente la precedencia entre dos aspects relacionados mediante herencia.

La regla que sigue AspectJ es que si dos aspects se encuentran relacionados por herencia el aspect derivado tiene mayor precedencia que el aspect base.

En estos casos de herencia no tendrá efecto una cláusula de "declare precedence".

Advices en un mismo aspect

Si en un mismo aspect hay varios advice que hacen referencia al mismo pointcut, no se aplican las reglas vistas anteriormente por estar en el mismo aspect. En estos casos el advice situado en primer lugar dentro del aspect es el que primero se ejecuta.

Opcional

AspectJ permite activar un advice cuando resulte cierta la comparación de valores en tiempo de ejecución que puedan ser evaluados al capturar un joinpoint. Esto se consigue utilizando la sentencia if(Expresión), cuya sintaxis es la siguiente:

Pointcut <nombre del pointcut>(parámetros) : if (expresión booleana);

Esta sentencia tiene dos características:

La sentecia if(Expresión) pointcut evalúa variables accesibles en tiempo de ejecución que resultarán en un resultado cierto o falso, lo cual decidirá si un joinpoint debe activar el correspondiente advice.

La Expresión puede estar compuesta de varios elementos lógicos, incluidos variables estáticas, contexto del joinpoint y otras declaraciones del pointcut.

Ejemplo:

```
Public aspect Ejemplo
{
        //variable para la comparacion
        private static final long salary = 2000l;
        pointcut ifJoinPointEjemplo( ) :
        if (thisJoinPoint.getThis( ) instance of MyClass       &&
           (MyClass) thisJoinPoint.getThis( )).getSalary( ) < salary &&
           ((MyClass) thisJoinPoint.getThis( )).getSalary( ) > 0) &&
```

```
            !withincode (* MyClass.get*( ));
            //advice after
            after( ) : ifJoinPointEjemplo ( ) && !within(Ejemplo +)
            {
               System.out.println("En el advice capturado por IfJoinPointEjemplo");
               System.out.println("Tipo join point: " + thisJoinPoint.getKind( ));
               System.out.println("Objeto: " + thisJoinPoint.getThis( ));
               System.out.println("Instancia MyClass " + ":" +
                     (MyClass) thisJoinPoint.getThis( ).getName( ) +
                     (MyClass) thisJoinPoint.getThis( ).getSalary( ));
               System.out.println("Signature: " + thisJoinPoint.getSignature( ));
               System.out.println("Linea codigo: " +
                     thisJointPoint.getSourceLocation ( ));
            }
      }
```

El advice after se activará cuando suceda lo siguiente:

- El objeto en ejecución sea del tipo MyClass
- El atributo salario del objeto sea menor que la constante salary
- El atributo salario del objeto sea mayor que cero
- El joinpoint actual no se encuentre dentro del método getSalary( ). Al usar el comodín, el joinpoint no debe estar en ningún método de la clase MyClass que empieze por get.

Las tres primeras condiciones son fácilmente entendibles. La última ha de ser incluida para evitar que la llamada al método getSalary( ) desde el advice se convierta en una llamada recursiva al advice y en definitiva, un bucle infinito.

Limitaciones

Como se ha visto, la precedencia que se puede indicar con AspectJ hace referencia a todos los advices dentro de un aspect, por lo tanto no se podrá indicar que un advice del aspect A se ejecute antes que otro del aspecto B, y que otro advice distinto del aspect B se ejecute antes que otro advice del aspect A. Esto supone una limitación del tejedor de AspectJ ya que la prioridad va a nivel de aspect, no de advice.

3.4 HERRAMIENTAS

Ante la importancia de la detección y resolución de conflictos, hay en desarrollo investigaciones y herramientas en el ámbito del desarrollo de sistemas orientados a aspects.

Vamos a entrar en más detalle de las siguientes herramientas:

ALPHEUS [1] **¡Error! No se encuentra el origen de la referencia.**, herramienta que ayuda al usuario en el desarrollo de aplicaciones orientadas a aspects.

Alpheus soporta un manejo de conflictos con la introducción de los planos y permitiendo al usuario definir conflictos potenciales entre aspects o planos, pudiendo especificar la política a seguir si se detecta un conflicto. Además, ofrece diferentes vistas del sistema a través de diagramas relacionando planos, conflictos y niveles de asociación, además de ofrecer diagramas UML.

SECRET [9] (SemantiC Reasoning Tool) implementa un modelo de detección de conflictos basada en los llamados Composition Filters (CF). Estos CF ofrecen una manera elegante para abordar los advices y la composición de éstos a través de especificaciones declarativas de los advice.

ASTOR [11], herramienta que consta de una serie de dispositivos que mejoran el tratamiento de conflictos en AspectJ. ASTOR consta de un administrador de conflictos, el cual detecta automáticamente conflictos y aplica estrategias de resolución más amplias que las que ofrece AspectJ. La detección de conflictos se basa en una clasificación de los mismos por niveles de semejanza y la resolución está basada en una taxonomía que ofrece seis categorías de resolución.

JECOM [22] (Java Extensible Conflict Manager), herramienta capaz de extraer interacciones entre aspects y clases, así como read-set y write-set de métodos y advices. Esta herramienta se basa en un motor de reglas, de manera que obtiene conflictos potenciales que se ajustan a las reglas que se han incluido en la base de conocimiento de dicho motor de reglas.

Existen otras herramientas orientadas a aspects como JBoss-AOP o SPRING-AOP. Sin embargo, estas herramientas no disponen de mecanismos para tratar los conflictos.

3.4.1 Alpheus

3.4.1.1 Arquitectura Reflexiva

Para entender la herramienta Alpheus vamos a explicar lo que se entiende por arquitectura reflexiva [12].

Una arquitectura reflexiva es aquella en la que los programas son capaces de manipular como datos alguna representación del estado de los mismos en tiempo de ejecución. Existen dos principios para esta manipulación: introspección (introspection) y realización (effectuation).

Introspección es la capacidad de un programa para observar y razonar sobre su propio estado. Se utiliza para encontrar qué metodos y propiedades están disponibles en las clases.

Realización es la capacidad de un programa para modificar dinámicamente su comportamiento sin necesidad de modificar el código.

Una arquitectura reflexiva está dividida en dos niveles:

- Nivel base: representa el propio sistema. Contiene objetos que resuelven una determinada funcionalidad y retornan información sobre el dominio que se está tratando en el sistema o sobre el dominio externo.
- Nivel meta: encargado de observar y/o modificar el comportamiento del nivel base. Está formado por meta-objetos. Este nivel devuelve información sobre el dominio interno del sistema y también sobre el nivel base.

Ambos niveles están conectados de forma que el nivel meta tiene acceso al nivel base pero el nivel base no sabe de la existencia del nivel meta.

Pueden existir varios niveles dentro del nivel meta, cada uno observando a los niveles inferiores. Y a su vez, puede haber varios planos dentro de cada nivel. El objetivo del plano es agrupar meta-objetos que satisfacen una determinada funcionalidad y tratarlos de forma conjunta, simplificando la manipulación y reutilización de los meta-objetos. Cuando se incorpora nueva funcionalidad a la aplicación hay que establecer una asociación entre el nivel base y el nivel metabase, es decir, entre el meta-objeto que incorpora la nueva funcionalidad y el objeto que la necesita.

El mecanismo de reflexión dependerá de las necesidades de la aplicación. Así, a veces se necesitará que todos los objetos tengan un meta-objeto asociado, y otras que solo algunos métodos tengan comportamiento reflexivo.

Se han identificado cuatro tipos para describir las posibles situaciones que nos podemos encontrar en una arquitectura reflexiva:

- Reflexión de clase: todos los objetos de una clase necesitan incorporar comportamiento adicional. Cada vez que un objeto de la clase recibe un mensaje, el mecanismo de reflexión redirige el flujo de control al nivel meta. Después, el control vuelve al nivel base.

- Reflexión de método: uno o varios métodos de una clase necesitan comportamiento adicional. Cuando se invoque un método afectado, el control pasará al nivel meta.

- Reflexión de objeto: si un objeto de una clase en particular necesita incorporar comportamiento adicional.

- Reflexión de método-objeto: se trata de una combinación de la reflexión de método y de la de objeto. Esto es, un método en particular de un objeto en concreto requiere comportamiento adicional.

Un objeto del nivel base puede estar asociado con uno o más meta-objetos del nivel meta. Si el comportamiento de cada meta-objeto es independiente del resto, no habrá problema. Pero podría haber conflicto si varios meta-objetos compiten por ser activados y el

comportamiento no es independiente, pudiendose producir inconsistencias o resultados no esperados.

Se creó una clasificación para distinguir los distintos niveles de conflictos:

- Meta-objeto meta-objeto: existe conflicto entres dos meta-objetos.
- Meta-objeto plano: si el conflicto existe entre un meta-objeto de un plano y todos los meta-objetos de otro plano.
- Plano plano: existe conflicto entre todos los meta-objetos de un plano y todos los meta-objetos de otro plano.
- Meta-objeto todos: este tipo de conflicto sucede entre un meta-objeto y el resto de meta-objetos existentes.

3.4.1.2 Descripción Alpheus

Alpheus [1] **¡Error! No se encuentra el origen de la referencia.** es una herramienta que ayuda al diseñador en la construcción de aplicaciones con características reflexivas, independizándole del propio framework y permitiéndole que se centre en las características de la propia aplicación.

Además, Alpheus tiene la ventaja de ser independiente del lenguaje, ya que el código se genera a partir de un módulo independiente.

Los sistemas reflexivos suelen tener limitaciones en cuanto a la separación funcional de meta-objetos. La herramienta Alpheus permite la creación de planos para agrupar meta-objetos según su funcionalidad. Además, el diseñador puede utilizar objetos y meta-objetos existentes, o crearlos mediante la herramienta. Al crear un objeto o meta-objeto con la herramienta se deben especificar los constructores, métodos e interfaces.

La arquitectura de Alpheus está compuesta de dos módulos principales como muestra la figura Figura 8: una Interfaz (Interface) y el Generador de Código (Program Code) encargado de generar código.

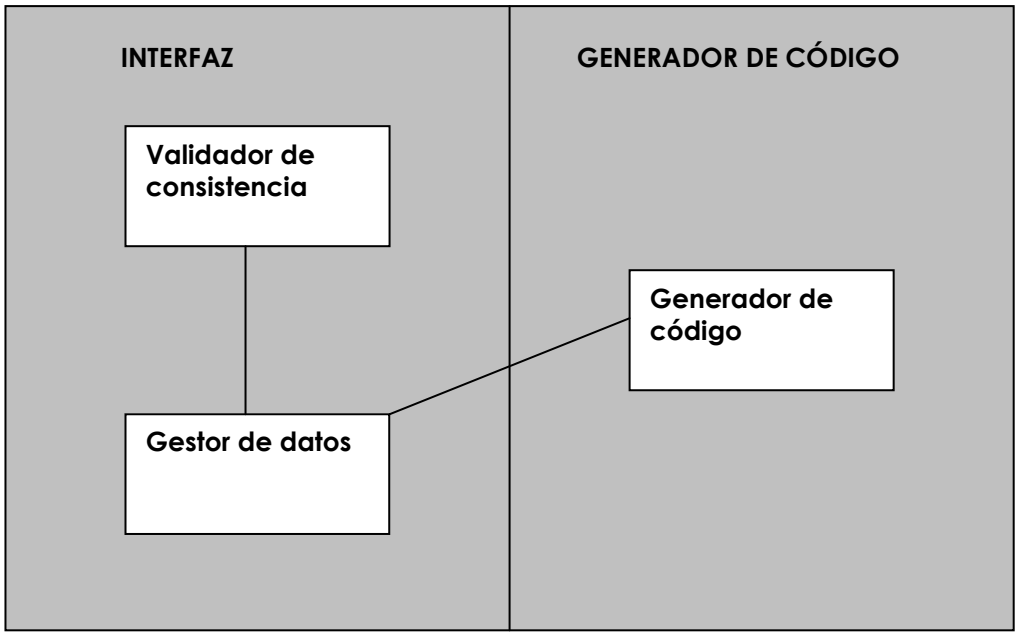


Figura 8. Arquitectura de Alpheus

La Interfaz (Interface) es utilizada por el diseñador para definir los componentes (objetos, meta-objetos, conflictos, asociaciones). Este módulo está formado por otros dos submódulos: Gestor de datos y Validador de consistencia.

El Gestor de datos (Data Manager) ofrece una interfaz para que el diseñador introduzca los datos, y después de validarlos los almacena.

Se guarda información relativa a los planos (nombre, objetos y meta-objetos incluidos en cada plano), asociaciones (nombre, objeto y meta-objeto, taxonomía, tiempo de invocación,etc), conflictos (nombre, participantes, taxonomía,etc), appliers (nombre, tipo, tiempo de invocación,etc) y la configuración particular de cada proyecto (directorio de trabajo, número máximo de planos, etc).

El Validador de consistencia (Consistence Validator) valida los datos que le pasa el Gestor de Datos.

La herramienta ayuda al diseñador a no cometer errores en la instanciación del framework, evitando que tenga acceso a datos u operaciones inválidas (por ejemplo, no permite definir una asociación cuya información se repita como característica en otra asociación, o no permite declarar un conflicto o una asociación entre dos meta-objetos del mismo plano).

El generador de código (Program Code) genera el código en base a la especificación del diseñador. No solo crea código, tambien genera ficheros de configuración necesarios para el framework y una guía de actividades para que el diseñador complete la instanciación del framework.

El código se genera a partir de la información almacenada por la Interfaz, y para permitir mayor flexibilidad cada componente tiene una clase encargada de generar el código que corresponde a su funcionalidad. Así, el componente Asociación (Association) tiene la clase AssociationGenerator para generar el código necesario para definir las asociaciones definidas.

Además, por simplicidad, los generadores de cada componente no se comunican entre sí, sino que lo hacen a través de un Repositorio (Repository).

3.4.2 Astor

Astor [13] es una herramienta que ayuda al desarrollador de aplicaciones AspectJ en el tratamiento de conflictos. La herramienta detecta automaticamente los conflictos y los resuelve de forma semiautomática mediante estrategias de resolución más avanzadas que las que dispone AspectJ.

La implementación de la herramienta está basada en el preprocesamiento del código AspectJ. Esto es, no reemplaza al tejedor de AspectJ, sino que mediante el preprocesamiento del código detecta automáticamente los conflictos y permite aplicar políticas de resolución semiautomáticamente.

La detección de conflictos se basa en una clasificación de conflictos por niveles de semejanza y la resolución se realiza según una taxonomía compuesta por seis categorías de resolución: en Orden, Orden Inverso, Opcional, Exclusivo, Nulo y Dependiente del Contexto (ya explicados anteriormente).

La herramienta está compuesta de tres componentes. La parte funcional consta de los componentes AProjectManager y AConflictManager, y el tercer componente es Interface, un

componente visual e interactivo por el cual el desarrollador codifica las clases, aspects e interfaces.

El componente AProjectManager gestiona los proyectos y las unidades de programas (métodos, atributos, puntos de corte,etc). Se genera una estructura que tiene como elemento principal el proyecto, y éste a su vez consta de componentes, los cuales pueden ser clases, aspects e interfaces. Además, una clase se compone de elementos de clase (métodos y/o atributos).

El componente AConflictManager se encarga de detectar y resolver los conflictos. La detección de conflictos se realiza mediante una distinción por niveles de semejanza o igualdad, basada en la estructura de los puntos de corte. Así, podemos distinguir:

- Conflictos de Semejanza Total: si dos aspects definen pointcuts, cuyos cortes primitivos y joinpoints coinciden, y además están asociados al mismo tipo de advice.

- Conflictos de Semejanza Parcial: si dos aspects definen pointcuts, cuyos joinpoints coinciden, y además están asociados al mismo tipo de advice.

En Astor los conflictos de semejanza total se representan por la clase AconflictTL y los de semejanza parcial por la clase AconflictPL.

Según se van generando las distintas unidades de programa en Astor, los pointcuts de los aspects se mapean en una tabla dinámica con información como aspect, pointcut, corte primitivo, joinpoint, advice, etc. Ya que los joinpoints pueden agruparse y abreviarse mediante el uso de comodines, puede ser que un pointcut produzca varias entradas en la tabla dinámica. El proceso de detección de conflictos consiste en analizar la tabla dinámica generada y crear objetos AConflict que el AConflictManager gestionará posteriormente.

La herramienta dispone de una representación matricial para visualizar los conflictos detectados. En esta matriz se sitúan los aspects en las filas y las columnas, y se representan los conflictos mediante botones situados en las celdas donde se cruzan los aspects que tienen conflictos (verdes si conflicto de semejanza parcial y rojos si son de semejanza total).

Además, la herramienta permite configurar el administrador de conflictos para analizar de forma aislada los conflictos según el nivel de semejanza (todos, conflictos de semejanza parcial, conflictos de semejanza total). También permite un filtrado de conflictos.

3.4.3 Secret

Para comprender esta herramienta vamos a explicar el concepto de Composition Filters (CFs), el cual nos permite razonar sobre los advices y la composición de éstos.

El modelo de CFs [9] [14] permite extender el modelo convencional orientado a objetos manipulando los mensajes enviados y recibidos. Además, los CFs fueron de los primeros mecanismos para abordar los aspects, sino el primero.

La siguiente figura (Figura 9) muestra la extensión mediante una abstracción del objeto con una capa que contiene filtros para manipular los mensajes enviados y recibidos. A su vez, los filtros se agrupan en 'filter modules' que proveen de contexto de ejecución a los filtros.

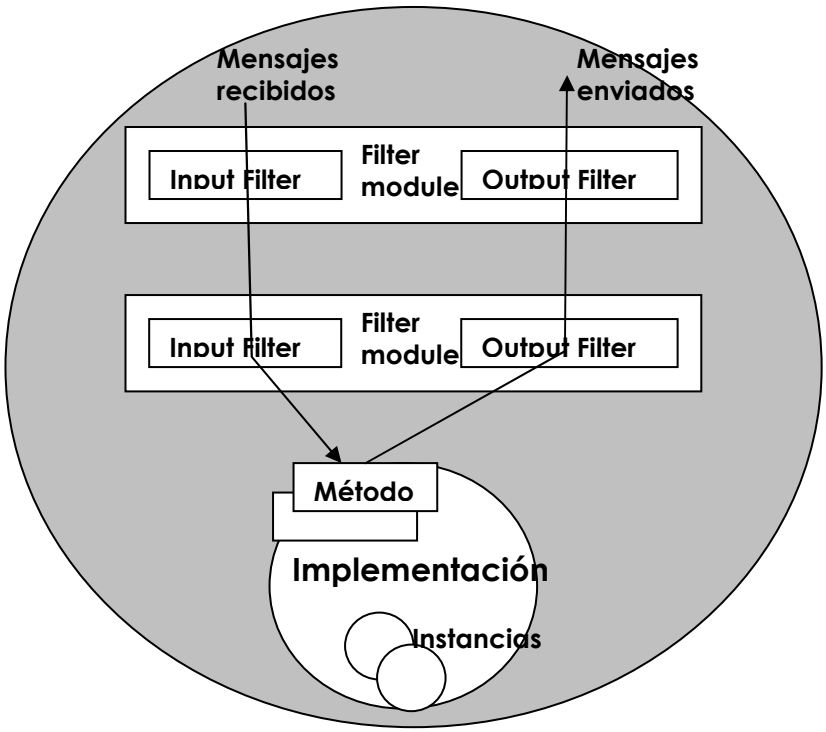


Figura 9. Esquema del modelo de CFs

Cada filtro especifica una manipulación de los mensajes, de forma que extiende los objetos convencionales. Los 'input filters' y los 'output filters' pueden manipular mensajes recibidos y enviados, respectivamente, por un objeto.

CFs permite una especificación declarativa de los aspects. Los advices en CFs son vistos como un conjunto de filtros, llamados filtermodule. Estos filtros se componen de los siguientes elementos:

nombre: el identificador del filtro

tipo: el tipo de filtro (Dispatch, Meta o user-defined)

condición: expresión booleana representando un estado, sobre el que el filtro se aplica

matching: indica sobre qué conjunto de mensajes este filtro es interesante

substitución: especifica qué propiedades son reemplazadas si ambas partes, matching y condición, tienen valor cierto. Se limita al target y al selector del mensaje.

Target: la parte objeto del mensaje.

Selector: la parte de método del mensaje.

Ejemplo de filtro:

Enqueue:Dispatch = {True =>[*.play] playlist.enqueue}

nombre tipo condición matching target selector

Hay varios tipos de filtros y para todos ellos el comportamiento se define por las acciones a realizar cuando el mensaje es rechazado o aceptado. Cada filtro tiene un tipo de filtro y un patrón de filtro. El tipo determina cómo manejar los mensajes después de ser comparados contra el patrón de filtro. Normalmente, los mensajes pasan secuencialmente a través de los filtros hasta que se dispachan. Se entiende por dispachar ejecutar un método o delegar el mensaje a otro objeto. Algunos de los tipos más utilizados son:

Dispatch: si el mensaje se acepta, se dispacha hacia el target específico del mensaje, en caso contrario, el mensaje continúa al siguiente filtro (si no hay más filtros se lanza una excepción)

- Error: si el filtro rechaza el mensaje, se levanta una excepción, en caso contrario, el mensaje continúa al siguiente filtro.

- Wait: si el mensaje se acepta, se continúa con el siguiente filtro. Sin embargo, el mensaje será encolado mientras que la evaluación del filtro resulte en rechazo.

- Meta: si el mensaje se acepta, el mensaje reified será enviado como parámetro de otro mensaje meta a un objeto. En caso contrario, el mensaje continúa con el siguiente filtro. El objeto que recibe el meta mensaje puede observar y manipular el mensaje, y luego reactivar su ejecución.

- Substitute: si el filtro acepta, ciertas propiedades del mensaje se podrán sustituir. Si el filtro rechaza, el mensaje continuará con el siguiente filtro.

Compose* es una implementación de Composition Filters (CFs) para la plataforma Microsoft .NET. Compose* está compuesto de muchos módulos, incluyendo un parser, herramientas de análisis estático y un weaver. La herramienta SemantiC Reasoning Tool (SECRET) implementa el modelo de conflictos.

Si múltiples filtermodules hacen referencia al mismo joinpoint se podrían producir conflictos. SECRET realiza un análisis estático sobre la semántica de los filtros y detecta posibles conflictos utilizando un modelo basado en una especificación XML adaptable por el usuario. En esta especificación XML se definen las acciones de aceptación y rechazo (accept y reject) de los filtermodules. Cada acción se especifica mediante una lista de operaciones sobre recursos. Además, también se especifican patrones para determinar la secuencia de operaciones permitidas sobre un recurso.

Cuando SECRET analiza un concern captura el orden de filtermodule y genera todas las posibles ejecuciones. Cada ejecución es una única combinación de acciones accept y reject de los filtros. Las operaciones de esas acciones proceden de la especificación XML y se realizan sobre los recursos. Luego se comprueban los patrones contra la secuencia de operaciones realizadas sobre los recursos. SECRET genera un listado de conflictos sobre un fichero HTML que muestra dónde y cómo suceden estos conflictos.

3.4.4 JECOM

JECOM [22] (Java Extensible Conflict Manager) es una herramienta cuyo objetivo es ayudar en el desarrollo y mantenimiento de un sistema orientado a aspects, analizando de forma automática el código y detectando los conflictos debido a interacciones entre aspects.

Esta herramienta se basa en un motor de reglas en el cual la información se recoge a través de hechos (facts) que forman la base de conocimiento del motor de reglas.

El análisis de las interacciones tiene cuatro fases secuenciales:

1. Análisis del Bytecode de la aplicación para extraer las interacciones entre aspects y clases directamente del bytecode generado por AspectJ.
2. Creación de la base de conocimiento (Knowledge Base), transformando la información mencionada en el punto 1 en un conjunto de hechos (facts) que formarán la base de conocimiento.

Tessier [20] ha desarrollado un modelo para representar interacciones entre aspects y clases en fases tempranas del diseño. Este modelo se compone de un grafo de nodos representando un joinpoint, por ejemplo, una interacción entre un aspect y una clase. En JECOM se extiende este modelo para representar la relación entre un método y un advice y entre advices. En este modelo extendido los elementos básicos son las variables compartidas, funciones (métodos y advices) y las interacciones. La interacción entre un aspect y una clase se compone de un conjunto de estructuras Link representando una interacción entre un advice y un joinpoint en una clase o en otro aspect. De la composición de todas las estructuras Link se obtiene un grafo que dependiendo de la aplicación puede llegar a tener un gran tamaño. Para facilitar el trabajo sobre los aspect de interés, se puede elegir un aspect desde el que empezar la actividad de ingeniería inversa.

Cada link se traduce en un hecho (fact) que se inserta en la base de conocimiento del motor de reglas. Sin embargo, este grafo no es suficiente para detectar todos los tipos de interacciones. Algunos se pueden producir por violar las condiciones que propone Bernstein.

Es posible evaluar si dos procesos pueden rodar en paralelo comprobando si los conjuntos de datos de input son independientes de otros conjuntos de datos de output, y si los conjuntos de datos de output son independientes. Bernstein [23] formalizó estas restricciones a través de tres condiciones expresadas utilizando la teoría de los set, teniendo en cuenta dos tipos de set: el Read-set como los datos leídos por un proceso, y el Write-set como los datos escritos por un proceso. Si Ra y Wa representan respectivamente el Read-set y el Write-set de un Advice a, y Rm y Wm representan respectivamente el Read-set y el Write-set de un método m, entonces las condiciones de Bernstein pueden escribirse como:

- $Ra \cap Wm = \emptyset$
- $Wa \cap Rm = \emptyset$
- $Wa \cap Wm = \emptyset$

Se deben cumplir las tres condiciones para garantizar la independencia entre un advice y un método.

En el modelo de JECOM se insertan en la base de conocimiento tublas RWS representando un acceso lectura-escritura (read-write access).

3. Creación de la base de reglas (Rule Base), utilizando el conjunto de reglas que representan las potenciales interacciones no deseadas.
4. Análisis de Conflictos, haciendo uso del motor de reglas se detecta si existen conflictos potenciales en la aplicación analizada consultando los datos recogidos. El motor de reglas continuamente aplica sentencias (reglas) del tipo si-entonces (if-then) a un grupo de afirmaciones (hechos o facts). JECOM además puede extenderse con nuevas reglas que el desarrollador considere oportunas. La base de conocimientos será actualizada cada vez que la aplicación orientada a aspectos se reconstruya.

JECOM se basa en un motor de reglas del tipo Jess [24], desarrollado en Java y que ofrece un API de Java para añadir hechos (facts) y reglas. También puede cargarse la base de conocimiento a partir de ficheros escritos en lenguaje Jess.

A continuación se muestra en la figura Figura 10 una regla escrita en lenguaje Jess que permite detectar advices recursivos.

```
(defrule AdviceSelfLoop
    (Link
            (aspectID ?A)
            (adviceID ?Adv)
            (adviceType ?AdvT)
            (declarationorder ?DO)
            (classID ?C)
            (methodID ?M)
            (joinpointID ?J)
    )
    (test (eq ?Adv ?M)

    (printout t  "Advice  '"  ?Adv "' advices itself ")
)
```

Figura 10. Regla para detectar un advice recursivo

4 EXPERIMENTACIÓN

4.1 PROPUESTA

Como hemos visto, existen varias herramientas para la detección de conflictos, aunque todas ellas están en período de desarrollo y no están disponibles.

La programación orientada a aspects ofrece muchas posibilidades para el desarrollador, pero puede llevar a comportamientos no deseados del sistema si la funcionalidad de varios joinpoints produce conflictos.

Nuestro objetivo es desarrollar un detector de conflictos que pueda utilizarse en cualquier sistema y que sea capaz de detectar conflictos producidos al acceder en modo escritura a alguno de los atributos de las clases contenidas en el sistema. Este detector corregirá algunos conflictos mediante precedencia de aspects. En otros casos, avisará al desarrollador de los conflictos y le facilitará medios para resolverlos.

El Detector de Conflictos desarrollado detecta y corrige los conflictos derivados del acceso a los atributos en la fase de prueba de la aplicación en la cual se ejecuta el detector. En las pruebas , el detector corregirá aquellos conflictos detectados y que no necesita del desarrollador para su solución. Para el resto de los conflictos detectados, el desarrollador deberá corregir el sistema para solucionar los conflictos, bien sea modificando la aplicación o con la ayuda de tablas incluidas en el Detector de Conflictos.

Una vez terminada la fase de pruebas, la aplicación se ejecutará sin conflictos utilizando la información obtenida en la fase de pruebas. Por lo tanto, las pruebas deberán ser exhaustivas para detectar todos los posibles conflictos que pudieran existir en el sistema.

4.2 MÓDULOS DEL DETECTOR DE CONFLICTOS

El Detector de Conflictos se compone de los siguientes módulos, de lo cuales, uno se trata de una base de datos Access, otro es un aspect escrito en AspectJ y los otros dos se corresponden con clases Java. Estos módulos serán añadidos a la aplicación sobre la que se quiere tener el detector de conflictos:

- Conflictos.mdb: base de datos Access
Esta base de datos contiene tablas que guardan información sobre los joinpoints, precedencias entre aspects e información sobre los conflictos entre aspects.

- DetectorJoinpoint: este módulo se corresponde a un aspect que captura los joinpoints necesarios para detectar los conflictos cuando:
    - Un aspect accede en modo escritura a un atributo de una clase y otro aspect o aspects acceden en modo lectura a ese mismo atributo. Ambos aspects se activarán al capturar el mismo joinpoint.
    - Varios aspects acceden en modo escritura a un atributo al capturar todos ellos el mismo joinpoint. Puede que haya varios otros aspects que accedan en modo lectura sobre ese mismo atributo.

- ComponenteJoinpoint: este módulo se trata de una clase, la cual se encarga de guardar en la base de datos Access (Conflictos.mdb) los joinpoints capturados por el detector.

- DetectorConflict: este módulo, que también se corresponde con una clase, accede a los detalles de los joinpoints almacenados en la base de datos Access por el módulo ComponenteJoinpoint. Después de estudiar los datos de la tabla, genera un aspect con las precedencias necesarias para evitar los conflictos que se pueden resolver, y una tabla con los conflictos que el desarrollador tendrá que resolver, al no ser capaz el detector de encontrar una solución.

4.2.1 Conflictos.mdb

La base de datos de tipo Access, Conflictos.mdb, se compone de cuatro tablas que guardan información utilizada para detectar y, si es posible, resolver los conflictos producidos entre aspects.

- La tabla 'Componentes' guarda información referente a los joinpoints capturados por el Detector. Esta tabla se compone de los siguientes campos:
    - Aspecto: nombre del aspect o clase desde el cual se lanza el joinpoint. Los aspect tendrán extensión .aj, mientras que

las clases tendrán extensión .java. Su valor se obtiene concatenando los métodos getSourceLocation y getFileName de la interfaz Joinpoint:

thisJoinPoint.getSourceLocation().getFileName().toString()

- Clase: nombre de la clase donde se ha ejecutado el joinpoint capturado. Este campo estará en blanco cuando se capture la ejecución de un advice. El valor de Clase se obtiene de los métodos getClass y getName: thisJoinPoint.getClass().getName()
- Instancia: referencia de la instancia del aspect asociado. Al ser un tipo de instanciación basado en control de flujo, los advice asociados al mismo pointcut tendrán la misma referencia. Su valor se obtiene ejecutando el método aspectOf asociado al aspect DetectorJoinpoint: DetectorJoinpoint.aspectOf().toString()
- InstanciaF:marca para indicar si es el comienzo o final del joinpoint. Contendrá los valores 'N' cuando se detecte el comienzo del joinpoint, y 'S' cuando se detecte el fin.
- JoinPoint: contiene el joinpoint capturado. Su valor se obtiene de los métodos getClass y getName: thisJoinPoint.getClass().getName().
- Localización: localización dentro del aspect o de la clase del joinpoint capturado. Su valor se obtiene del método getSourceLocation:
  thisJoinPoint.getSourceLocation()
- ReferenciaObj: referencia del objeto implicado en el joinpoint. Este campo estará en blanco cuando se capture la ejecución de un advice.Su valor se obtiene a través de los métodos getTarget y hashCode:

  thisJoinPoint.getTarget().hashCode()
- TipoAdvice: cuando se capture la ejecución de un advice, indicará si el advice es de tipo 'before', 'after' o 'around'. Para obtener el tipo del advice se accederá a la variable

thisJoinPoint, la cual contendrá el joinpoint y el tipo de advice.

- TipoCorte: tipo de joinpoint, 'method-call', 'constructor-call', 'field-get', 'field-set'…. Este campo estará en blanco cuando se capture la ejecución de un advice. El valor de este campo se obtiene del método getKind: thisJoinPoint.getKind().
- JoinPointPadre: joinpoint dentro del cual se ha ejecutado el joinpoint capturado. Este campo contendrá valor para los joinpoint capturados durante la ejecución de un advice. Su valor se obtendrá al crear el registro en la tabla Componentes cuando se detecte el fin de un joinpoint
- Conflicto: indicará si el joinpoint capturado está afectado por algún conflicto producido por el advice asociado u otro advice que ejecute el mismo joinpoint. Después de ejecutar las pruebas, se revisarán los datos recogidos en la tabla de Componentes para detectar conflictos. Si en este proceso se detecta conflicto entre aspects, este campo se actualizará con el valor 'S' en todos los registros de los joinpoints afectados por tal conflicto.
- Id: número generado automáticamente para saber si un joinpoint ha sido ejecutado con anterioridad a otro.

- La tabla ConflictosCab contiene información asociada a los conflictos detectados y que pueden ser resueltos con información adicional que debe proporcionar el desarrollador.
Los campos que contiene esta tabla son:

- JoinPoint: joinpoint capturado y que forma parte de un conflicto
- TipoAdvice: tipo del advice afectado por el joinpoint (before, after, around).
- Advice: identificación del advice afectado por el joinpoint. Esta identificación consiste en el nombre del aspect y la línea dentro del aspecto donde empieza el advice.

- Revisado: identificador (S/N) actualizado por el desarrollador para indicar que este conflicto ha sido revisado y se puede utilizar la información contenida en esta tabla para tomar acciones en tiempo de ejecución.
- PermitirEjecución: identificador (S/N) actualizado por el desarrollador para indicar si se puede ejecutar el advice al que hace referencia el registro, en caso de ser activado en tiempo de ejecución de la aplicación, y haberse ejecutado ya otro advice asociado al mismo joinpoint. El primer advice ejecutado siempre podrá ejecutarse, puesto que el desarrollador habrá dado un orden de precedencia a los advice involucrados en el conflicto.

▪ La tabla Precedencias guarda información sobre precedencias entre pares de aspects para evitar conflictos. Posteriormente, esta tabla se utilizará para crear un aspect de precedencias con todas las precedencias contenidas en dicha tabla.
Los campos de esta tabla son:

- Id: clave para identificar univocamente a la precedencia. El valor de este campo se asigna automáticamente.
- JoinPoint: joinpoint capturado y que forma parte de un conflicto.
- AspectoPrimero: aspect que debe ejecutarse en primer lugar para evitar conflicto
- AspectoSegundo: aspect que debe ejecutarse en segundo lugar para evitar conflicto

4.2.2 DetectorJoinpoint

DetectorJoinpoint es un aspect instanciado en base a control de flujo. Esto es, se creará una nueva instancia del aspect por cada nuevo flujo de control del conjunto de joinpoints definidos como parámetro en la declaración de instanciación percflow(conflictsPointcuts). Este tipo de instanciación nos va a resultar de utilidad para conocer cuando

empieza un joinpoint y cuando termina, y poder saber si otros joinpoints se ejecutaron entre el comienzo y el final de ese primer joinpoint.

Los joinpoints que va a capturar este aspect se definen dentro del pointcut conflictPointcuts y son los siguientes:

- adviceexecutionX: es un pointcut del tipo adviceexecution, el cual captura la ejecución de los advice.
    - Hay tres advice asociados a este pointcut. Uno se ejecutará antes del joinpoint, otro después y el otro mientras se ejecuta .
    - Los advice que se ejecutan antes y después del joinpoint se ejecutan en tiempo de pruebas y guardarán los detalles del joinpoint en la tabla de Access 'Componentes', con la diferencia de que el advice que se ejecuta antes guardará una marca en la tabla indicando inicio de joinpoint, mientras que el que se ejecuta después guardará la marca de fin de joinpoint.
    - Para obtener parte de los datos almacenados en la tabla Componentes, el método getParameters (JoinPoint jp) accede a los parámetros del joinpoint capturado para conseguir el nombre de dicho joinpoint y el nombre de la clase donde se encuentra el joinpoint.
      El resto de datos se obtiene de métodos asociados a thisJoinPoint.
    - El advice que se ejecuta durante la ejecución del advice capturado se ejecutará unicamente en tiempo de ejecución, y gracias a la información recogida en las tablas de Access en el tiempo de pruebas, ejecutará o no el advice dependiendo de si el joinpoint y el advice forman parte de un conflicto y si el desarrollador creyó oportuno que se ejecutara si se produjese el conflicto.
- callPointcut: es un pointcut del tipo call, que captura todas las llamadas a métodos, excluyendo las llamadas a métodos del paquete java a través del pointcut callsToBeExcluded.
  pointcut callsToBeExcluded(): call (* java.*.*.*(..));
El tipo de corte asociado al pointcut callPointcut es before y after. En

ambos caso, el advice asociado guardará los detalles del joinpoint capturado en la tabla 'Componentes'.

- newPointcut: es un pointcut de tipo new, el cual captura todas las llamadas a constructores de clase. El tipo de corte asociado al pointcut newPointcut es before y after. En ambos caso, el advice asociado guardará los detalles del joinpoint capturado en la tabla 'Componentes'.

- setPointcut: se trata de un pointcut del tipo set, el cual captura todos los accesos a atributos en modo escritura, siempre y cuando estos accesos no estén en el propio aspect DetectorJoinpoint o en la clase ComponenteJoinpoint, evitando capturar los accesos a las variables utilizadas para la detección de conflictos. El tipo de corte asociado al pointcut setPointcut es after. El advice asociado guardará los detalles del joinpoint capturado en la tabla 'Componentes'.

- getPointcut: es un pointcut del tipo get, el cual captura todos los accesos a atributos en modo lectura, excluyendo aquellos accesos incluidos en el aspect DetectorJoinpoint o en la clase ComponenteJoinpoint. El tipo de corte asociado al pointcut getPointcut es after. El advice asociado guardará los detalles del joinpoint capturado en la tabla 'Componentes'.

4.2.3 ComponenteJoinpoint

La clase ComponenteJoinpoint se encarga de manipular objetos de esta clase, los cuales, además, almacenan información de joinpoints capturados en una base de datos que luega servirá para detectar los conflictos entre aspects.

Esta clase tiene dos constructores. Uno de ellos crea el objeto a la vez que guarda los datos en la tabla Access 'Componentes' de la base de datos Conflictos.mdb.

public ComponenteJoinpoint (String j, String t, String c, String l, String i, String cl, String a, int r, String o, String f)

El otro constructor simplemente crea el objeto asociado a la clase.

public ComponenteJoinpoint (String j, String t, String c, String l, String i, String cl, String a, int r, String o, String f, int id)

Los métodos de la clase ComponenteJoinpoint son:

- Métodos para acceder a los atributos de la clase:
    - public String getJoinPointStr (): devuelve un String con la expresión del joinpoint (JoinPointStr).
    - public void setJoinPointStr (String s): da valor a JoinPointStr con el String pasado como parámetro.
    - public String getTipoAdvice (): devuelve un String con el tipo de advice del joinpoint (TipoAdvice).
    - public void setTipoAdvice (String t): da valor a TipoAdvice con el String pasado como parámetro.
    - public String getTipoCorte (): devuelve un String con el tipo de corte del joinpoint (TipoCorte).
    - public void setTipoCorte (String t): da valor a TipoCorte con el String pasado como parámetro.
    - public String getLocalizacion (): devuelve un String con la localización dentro del aspect donde se localiza el joinpoint (Localizacion).
    - public void setLocalizacion (String l): da valor a Localizacion con el String pasado como parámetro.
    - public String getInstancia (): devuelve un String con el valor de la instancia del aspect (Instancia).
    - public void setInstancia (String i): da valor a Instancia con el String pasado como parámetro.
    - public String getClase(): devuelve un String con el nombre de la clase que lanzó el joinpoint (Clase).
    - public void setClase (String c): da valor a Clase con el String pasado como parámetro.
    - public String getAspecto (): devuelve un String con el nombre del aspecto donde se encuentra el pointcut asociado al joinpoint (Aspecto).

- public void setAspecto (String a): da valor a Aspecto con el String pasado como parámetro.
- public String getObjeto (): devuelve un String con la referencia del objeto afectado por el joinpoint (Objeto).
- public void setObjeto (String o): da valor a Objeto con el String pasado como parámetro.

▪ Métodos para acceder a la información almacenada en la base de datos Conflictos.mdb:

- public int guardarComponente (): guarda en la tabla Componentes de la base de datos Access Conflictos.mdb los datos del joinpoint que se encuentran en el objeto this de la clase. Este método devuelve un valor distinto de cero si hubo algún error. Después de guardar los datos del joinpoint, se llama al método detectar_conflictos.
- int detectar_conflictos (): comprueba si el joinpoint está involucrado en un conflicto. Si los datos corresponden al final del joinpoint, se obtiene el Id del inicio y fin del joinpoint mediante los métodos obtenerInicio y obtenerFin. A continuación, se distingue entre un joinpoint dentro de un aspect o de una clase comprobando la extensión del campo Aspecto (.aj para aspect). Si se trata de un aspect, se actualizarán los datos de dicho joinpoint mediante el método actualizarDatosJoinpoint. Si se trata de una clase, se comprobará si hay conflicto mediante el método comprobarGetSet.
- int obtenerInicio (): devuelve un Int con el valor del campo Id que corresponde al registro de la misma instancia pero con el campo InstanciaF a 'N'.
- int obtenerFin (): devuelve un Int con el valor del campo Id que corresponde al registro de la misma instancia pero con el campo InstanciaF a 'S'.
- int actualizarDatosJoinpoint (int inicio, int fin): actualiza el campo JoinPointPadre de los registros de la tabla Componentes cuyo Id está comprendido entre los valores

inicio y fin pasados por parámetro. El valor asignado al campo JoinPointPadre es el de la expresión del joinpoint capturado. Este campo servirá para saber qué joinpoint originó un joinpoint set o get en concreto (dato que no lo tenemos en el momentode grabar el registro en la tabla).

- int comprobarGetSet (int inicio, int fin): lee los registros cuyo tipo de corte es 'field-set' o 'field-get' y llama al procedimiento detectarConflictosGetSet para ver si están involucrados en algún conflicto.
- int detectarConflictosGetSet (int inicio, int fin, String joinPointGetSet): comprueba si en la tabla Componentes hay más de un registro entre los registros cuyo Id están entre inicio y fin, que hacen referencia al atributo indicado en joinPointGetSet, cuyo tipo de corte es 'field-set' y hacen referencia al mismo objeto. Esto es, comprueban si hay más de un joinpoint que intenta acceder en modo escritura al mismo atributo del mismo objeto. Si hay más de un registro que acceda en modo escritura, se actualizarán los registros que acceden al mismo atributo, ya sea en modo escritura o en modo lectura para indicar que existe conflicto (actualizar campo Conflicto con valor 'S'). Si solo hubiera un registro que acceda en modo escritura, y existen uno o varios joinpoints que accedan en modo lectura, se llamará al procedimento guardarPrecedencias para dar precedencia al aspect con el acceso en modo escritura, y de este modo, los otros aspect cuando accedan en modo lectura, leerán el valor ya modificado.
- int guardarPrecedencias (int incio, int fin, String joinPointGetSet, String aspectSet): lee de la tabla Componentes aquellos registros que hacen referencia al atributo indicado en joinPointGetSet en modo lectura (tipo de core igual a 'field-get'), insertando un nuevo registro en la tabla Precedencias, indicando como primer aspect y más prioritario (AspectoPrimero) el aspect que contiene el acceso en modo escritura (aspectSet), y como segundo aspect (AspectoSegundo), el aspect leído de la tabla

Componentes, que accede en modo lectura al mismo atributo.

- public int inicializarComponentes (): este método es llamado al inicio de la aplicación en modo pruebas para inicializar la tabla Componentes, y dejarla lista para ser actualizada en la próxima ejecución de la aplicación. También se inicializará la tabla Precedencias.

4.2.4 DetectorConflict

La clase DetectorConflict revisa los datos almacenados en la tabla Componentes y detecta los conflictos entre aspects. De esta forma el desarrollador podrá tomar las acciones oportunas para resolver los conflictos.

Además, partiendo de los datos almacenados en la fase de pruebas en la tabla Precedencias, genera un fichero, el cual se corresponde a un aspect con las precedencias necesarias para evitar algunos los conflictos producidos cuando sobre el mismo joinpoint y objeto, un aspect accede a un atributo en modo escritura, y uno o varios aspects acceden al mismo atributo en modo lectura.

La clase DetectorConflict se compone de los siguientes métodos:

- int detectarConflictos (): lee los distintos joinpoints de la tabla Componentes que están involucrados en un conflicto (campo Conflicto con valor 'S') e inserta en la tabla ConflictosCab los datos de joinpoint, el nombre del aspect, su localización dentro del aspect y el tipo de advice correspondiente. Esta tabla será revisada por el desarrollador para tratar de resolver los conflictos. El método detectarConflictos deberá ser añadido al final de la batería de pruebas de la aplicación de la que se desea detectar los conflictos.

- int grabarFicheroPrecedencias (): genera un aspect con las precedencias necesarias para solucionar los conflictos producidos cuando un atributo es accedido en modo escritura y otros aspect acceden en modo lectura al mismo atributo. Para ello, se recorre la tabla Precedencias y genera un fichero con la estructura de un aspect, y tantas líneas con sentencias 'declare precedence' como pares de aspects haya en la tabla Precedencias. Este procedimiento deberá ser

modificado por el desarrollador para determinar la ubicación del fichero de precedencias.

- int comprobarConflicto (String joinpointStr, String advice, String tipoAdvice):

consulta la tablas ConflictosCab y chequea si el advice se puede ejecutar o no. Si el advice está referenciado en esta tabla, solo podrá ejecutarse si el conflictos ha sido revisado y se permite su ejecución mediante los campos Revisado y PermitirEjecución, respectivamente.

4.3 CASO PRÁCTICO: TELECOM

El programa telecom distribuido con AspectJ simula conexiones telefónicas.

La aplicación tiene tres funcionalidades principales: customers (clientes), calls (llamadas) y connections (conexiones).

La aplicación Telecom incluye dos aspect, el aspect Timing que añade funcionalidad para registrar el tiempo de conexión telefónica utilizando un timer en cada conexión. El otro aspect, Billing, utiliza el tiempo de conexión para cobrar al usuario que inició la llamada.

4.3.1 Requisitos Software

A continuación se detallan los requisitos software de la aplicación telecom.

Primero describimos los requisitos funcionales.

IDENTIFICADOR	DESCRIPCIÓN
RS001	Cada llamada estará identificada por un usuario que llama (caller) y otro usuario llamado (receiver)
RS002	Al crearse una llamada se creará una conexión de tipo local o de larga distancia según las áreas de los usuarios sean iguales o distintas, respectivamente. La conexión se creará en estado pendiente de ser aceptada por el llamado
RS003	Una llamada contendrá un vector de conexiones
RS004	Cuando el usuario llamado acepta la llamada la conexión pasa a estado completo
RS005	Cuando uno de los usuarios asociados a una llamada cuelga la llamada, se eliminan todas las conexiones asociadas a la llamada

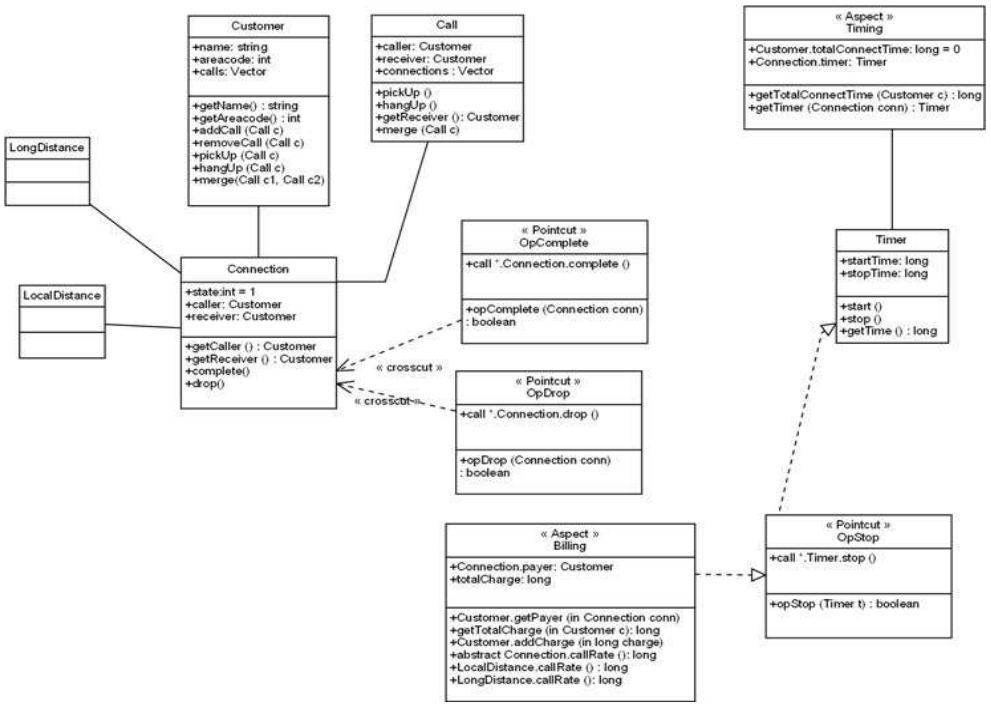
RS006	Si se elige unir las conexiones de una llamada, se eliminarán las conexiones de dicha llamada y se añadirán a la llamada actual
RS007	Un usuario se identifica mediante un nombre, un área y un vector de llamadas
RS008	Si un usuario realiza una llamada, creará una llamada y la añadirá a dicho usuario
RS009	Si un usuario accepta una llamada, hará que la conexión asociada se complete y añadirá la llamada al usuario
RS010	Si un usuario une dos llamadas, hará que las conexiones de ambas llamadas se unan en la primera llamada, y la segunda llamada se elimine
RS011	Si un usuario cuelga una llamada, hará que se eliminen las conexiones de la llamada y que se elimine la llamada asociada al usuario
RS012	Al crearse una conexión se asocia a la conexión que el usuario que llama será el usuario que pague la llamada
RS013	Al eliminar una conexión se calculará el coste de la llamada según el tiempo de conexión. El cálculo se realizará multiplicando el tiempo de la conexión por el coste según sea la conexión local o de larga distancia.

También se han definido los siguientes requisitos no funcionales.

IDENTIFICADOR	DESCRIPCIÓN
RS101	Al crearse una conexión se iniciará un temporizador para calcular el inicio de la conexión
RS102	Cuando se elimine una conexión se parará el temporizador para calcular el fin de la conexión y poder calcular la duración de la conexión
RS103	Un temporizador se identificará por un tiempo de inicio y un tiempo de fin de llamada
RS104	Habrá un log para conocer la duración de la llamada

4.3.2 Diseño

A continuación se muestra el gráfico del diseño de la aplicación Telecom.



Los requerimientos de tiempo y facturación de las conexiones se tratan de crosscutting concerns. Por lo tanto, éstos serán implementados mediante aspects.

El aspect Timing controla el tiempo total de cada cliente y se añade como un timer a cada conexión.

El aspect Billing calcula el coste de cada conexión y se basa en la información de tiempo obtenida del aspect Timing.

El aspect Timing declara una variable de la clase Connection (Connection.timer) y otra variable de Customer (Customer.totalConnectTime), las cuales solo el aspect Timing puede ver.

El aspect Timing registra la duración de cada conexión y asocia el tiempo total de conexión con cada cliente. Además, extiende la estructura de las clases Customer y Connection, interceptando la ejecución de los métodos Connection.complete() y Connection.drop().

Se añadió el aspect TimerLog para guardar un log de cuándo un timer se empieza y para. Este aspect intercepta la ejecución de los métodos Timer.start() y Timer.stop().

El aspect Billing trata la funcionalidad de facturación de las conexiones basándose en la información del aspect Timing. Declara nuevas funciones a las clases Customer y Connection para guardar el cliente que pagará la llamada, y para facturar de distinta forma las llamadas de corta y larga duración. Este aspect intercepta cuando se ejecuta el constructor de la clase Connection para identificar el que llama (caller), que será el que pague la llamada.

4.3.3 Ampliación Telecom

A la aplicación Telecom se le han añadido los siguientes requisitos software funcionales:

IDENTIFICADOR	DESCRIPCIÓN
RS301	Si el usuario que llama es Jim, se cambiará su área al del usuario llamado para que todas las llamadas de

	Jim sean locales
RS302	Si el usuario llamado es Crista, se cambiará el pagador de la conexión a Crista
RS303	Se añadirá un atributo de puntos a los usuarios
RS304	Se sumará un punto al usuario que llama cuando se termine la conexión
RS305	Se sumará cuatro puntos extra al usuario que llama si la llamada se realiza entre los días 10 y 20 de cada mes

Como requisitos no funcionales se han añadido a la aplicación telecom los siguientes:

IDENTIFICADOR	**DESCRIPCIÓN**
RS401	Mostrar un log de las llamadas
RS402	Flexibilidad para cambiar la aplicación ya existente
RS403	Evitar conflictos de funcionamiento entre los distintos módulos

Para cumplir con estos nuevos requisitos se han añadido una serie de aspects. Además, al añadir la nueva funcinalidad a través de aspects se cumple con el requisito no funcional de flexibilidad para cambiar la aplicación telecom que ya existía (no se modifican los módulos de la aplicación general). Y para cumplir con el requisito no funcional de evitar conflictos entre los distintos módulos, se utiliará el Dectector de Conflictos desarrollado en este proyecto.

Se ha añadido el atributo 'puntos' a la clase Customer, para posteriormente utilizarlo en dos de los aspects añadidos.

A continuación, explicamos los aspect añadidos:

- Aspect DesvioLlamada: este aspect cambia el área asociado al usuario que llama. En particular, si el usuario que llama es "Jim", cambia el área al mismo que el del usuario llamado. De esta forma, todas las llamadas que haga Jim serán locales.

    - Joinpoint: call(Call.new(Customer,Customer))
    - Pointcut:
      pointcut desvioLlamada(Customer caller, Customer rec) : call(Call.new(Customer,Customer)) && args(caller,rec);
    - Advice: el advice es de tipo before y está asociado al pointcut desvioLlamada

```
before (Customer caller, Customer rec):
desvioLlamada(caller,rec){      JoinPoint j=thisJoinPoint;
        System.err.println("Desvio Llamada para
"+caller.getName()+" de      area "+caller.getAreacode()+"
a area "+rec.getAreacode());
        if (caller.getAreacode() != rec.getAreacode() &&
caller.getName()                =="Jim"){
           caller.setAreacode(rec.getAreacode());
        }
}
```

- AvisoLlamadas: este aspect avisa del área del usuario que llama, así como del usuario llamado.

    - Joinpoint: call(Call.new(Customer,Customer))
    - Pointcut:
      pointcut darAvisoLlamada(Customer caller, Customer rec) : call(Call.new(Customer,Customer)) && args(caller,rec);
    - Advice: el advice es de tipo before y está asociado al pointcut darAvisoLlamada.

```
before (Customer caller, Customer rec):
darAvisoLlamada(caller,rec){
        JoinPoint j=thisJoinPoint;
        System.err.println("Atencion, Llamada desde área " +
                caller.getAreacode()+ " a area
```

```
"+rec.getAreacode());
}
```

- CobroCentralizado: este aspect cambia el pagador a un nuevo usuario que hace las veces de central de pago. En el mismo aspect se crea este nuevo usuario de forma transparente al resto del sistema.

  - Joinpoint: call(Connection+.new(..))
  - Declaración adicional: se declara un nuevo usuario que se corresponde a la central de pago.

    ```
    Customer cent = new Customer("Cent", 650);
    ```

  - Advice:

    ```
    after(Customer caller, Customer rec) returning (Connection
    conn):
        args(caller, rec) && call(Connection+.new(..)) {
            JoinPoint j=thisJoinPoint;
            if (caller.getName()== "Jim"){
                    System.err.println("Cobro Centralizado");
                    conn.payer=cent;
            }
    }
    ```

- CobroRevertido: este aspect cambia el pagador al usuario que es llamado en vez de ser el que realiza la llamada.

  - Joinpoint: call(Connection+.new(..))
  - Advice:

    ```
    after(Customer caller, Customer rec) returning (Connection
    conn):args(caller, rec) && call(Connection+.new(..)) {
            JoinPoint j=thisJoinPoint;
            System.out.println("En CobroRevertido");
            if (rec.getName()== "Crista"){
                    System.err.println("Cobro Revertido a
    +rec.getName());
                    conn.payer=rec;
            }
    }
    ```

- SumaPuntos: este aspect aumenta en uno el número de puntos del usuario que llama.
    - Joinpoint: call(void Connection.drop());
    - Pointcut: pointcut sumaPuntos(Connection c): target(c) && call(void Connection.drop());
    - Advice:

```
after(Connection c): sumaPuntos(c) {
   JoinPoint j=thisJoinPoint;
   System.err.println("Suma de puntos");
   int puntos=c.getCaller().getPuntos();
   c.getCaller().setPuntos(puntos+1);
}
```

- DiasFelices: este aspect aumenta el número de puntos del usuario que llama en 4 si la llamada se realiza los primeros diez días del mes.
    - Joinpoint: call(void Connection.drop());
    - Pointcut: pointcut extraPuntos(Connection c): target(c) && call(void Connection.drop());
    - Advice:

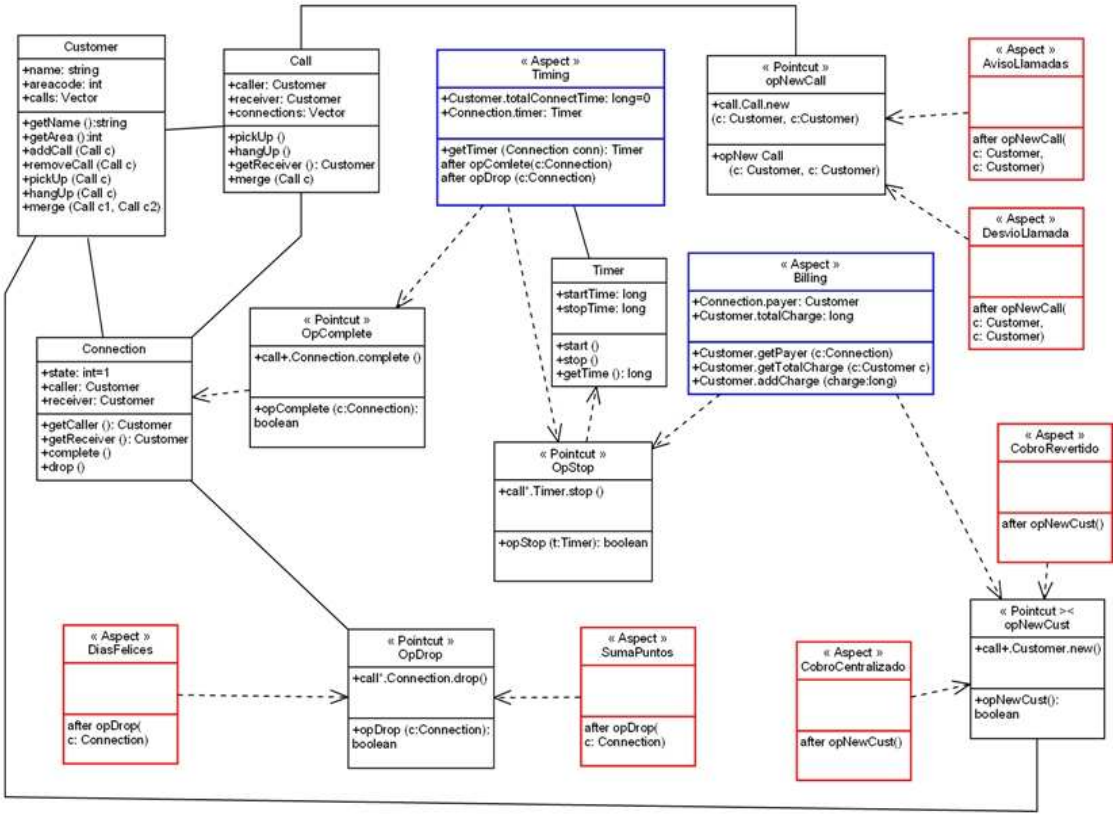
```
after(Connection c): extraPuntos(c) {
   int dia;

   JoinPoint j=thisJoinPoint;
   Calendar cal=Calendar.getInstance();
   dia=cal.get(Calendar.DAY_OF_MONTH);
   if (dia >= 1 && dia <= 10){
         System.err.println("Suma de puntos");
         int puntos=c.getCaller().getPuntos();
                              c.getCaller().setPuntos(puntos+4);
   }
}
```

- Tracing: este aspect captura todas las llamadas de la clase Llamdas.
    - Joinpoint: call(* telecom.Call.*(..));
    - Pointcut: pointcut TraceAll() : call(* telecom.Call.*(..));

- Advice:

```
before () : TraceAll(){
    JoinPoint j=thisJoinPoint;
    System.err.println("Trace: "+thisJoinPoint.getSignature());
}
```

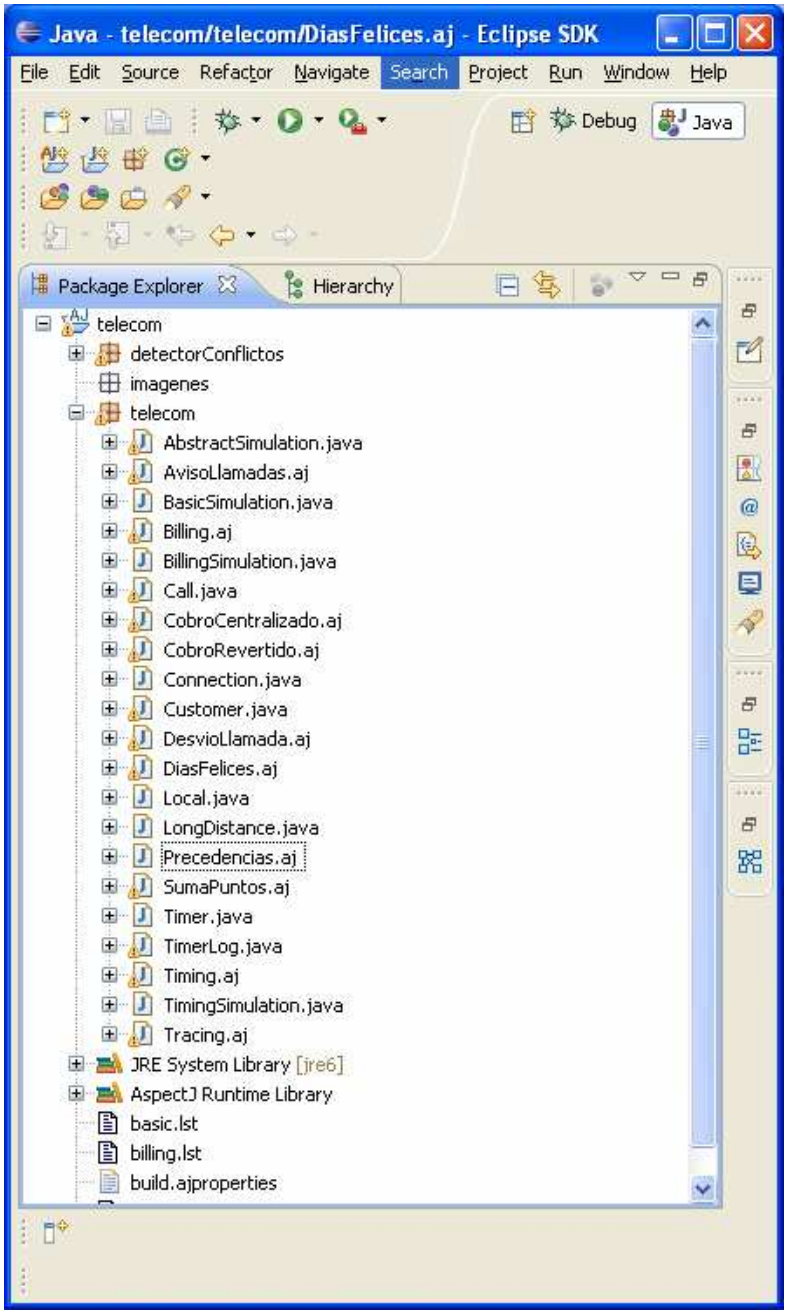
El siguiente gráfico muestra el diseño de la aplicación Telecom una vez añadidos los nuevos aspects de los requerimientos añadidos.



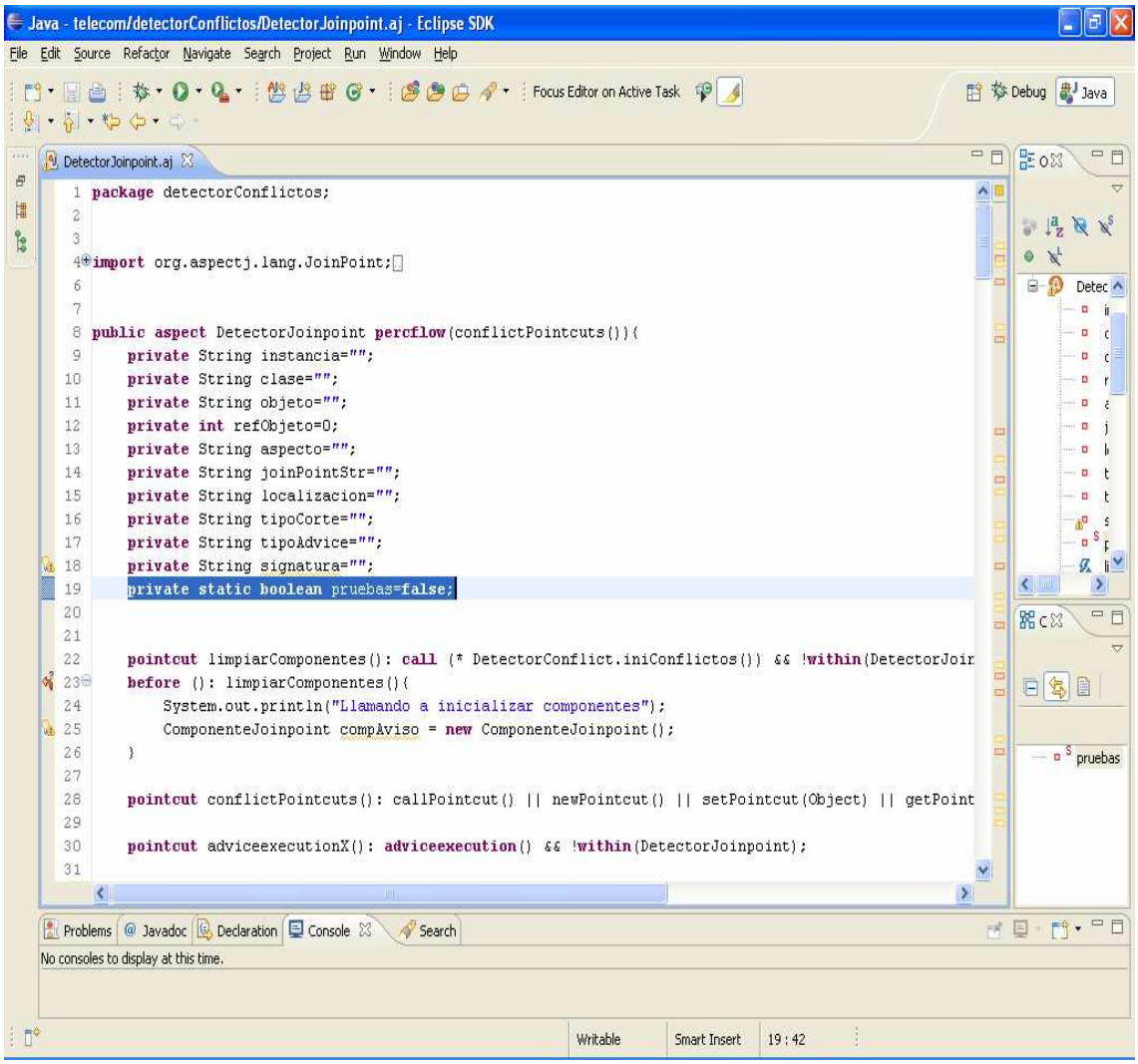
4.3.4 Conflictos en Telecom

Para detectar los conflictos entre aspects dentro de la aplicación Telecom utilizando el detector de conflictos desarrollado, se procede de acuerdo a los siguientes pasos:

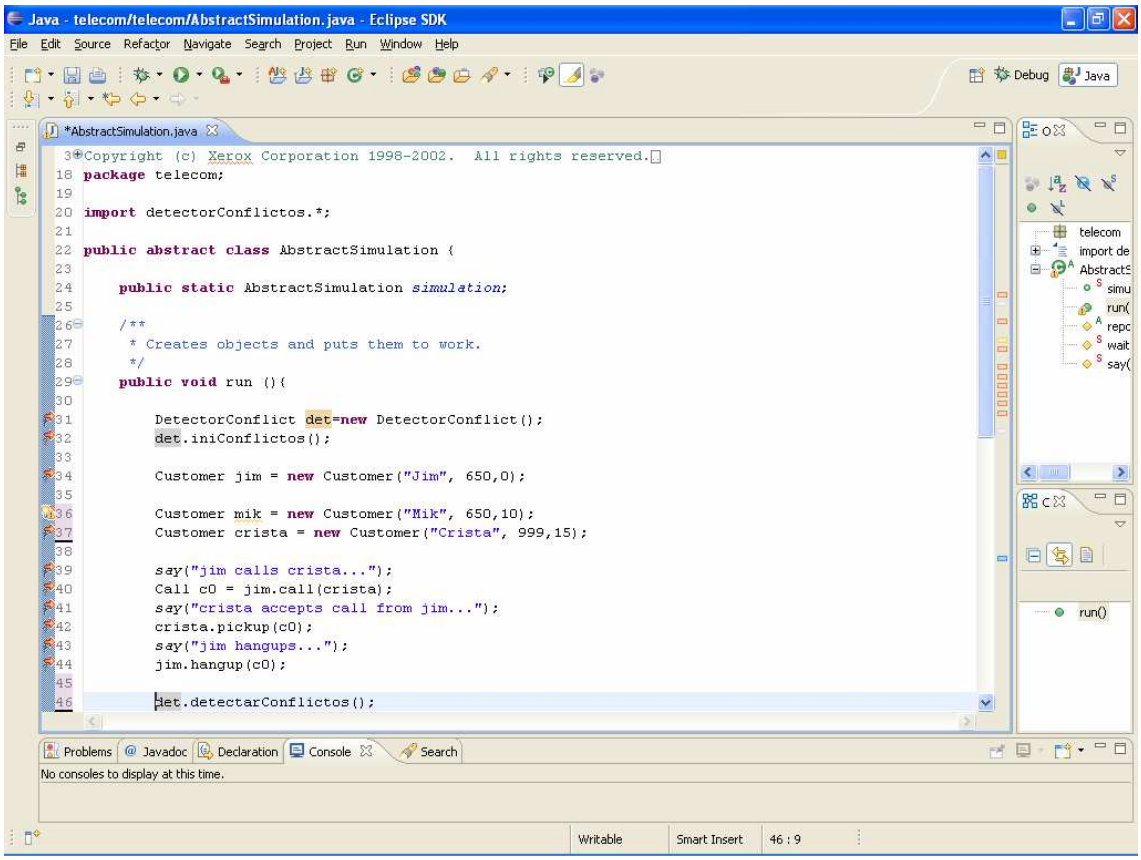
1. Añadir el paquete detectorConflictos a la aplicación Telecom, incluyendo ComponeneteJoinpoint.java, DetectorConflict.java y DetectorJoinpoint.aj, quedando la estructura de la aplicación de la siguiente forma:


Java - telecom/telecom/DiasFelices.aj - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Java
Package Explorer
Hierarchy
telecom
detectorConflictos
imagenes
telecom
AbstractSimulation.java
AvisoLlamadas.aj
BasicSimulation.java
Billing.aj
BillingSimulation.java
Call.java
CobroCentralizado.aj
CobroRevertido.aj
Connection.java
Customer.java
DesvioLlamada.aj
DiasFelices.aj
Local.java
LongDistance.java
Precedencias.aj
SumaPuntos.aj
Timer.java
TimerLog.java
Timing.aj
TimingSimulation.java
Tracing.aj
JRE System Library [jre6]
AspectJ Runtime Library
basic.lst
billing.lst
build.ajproperties

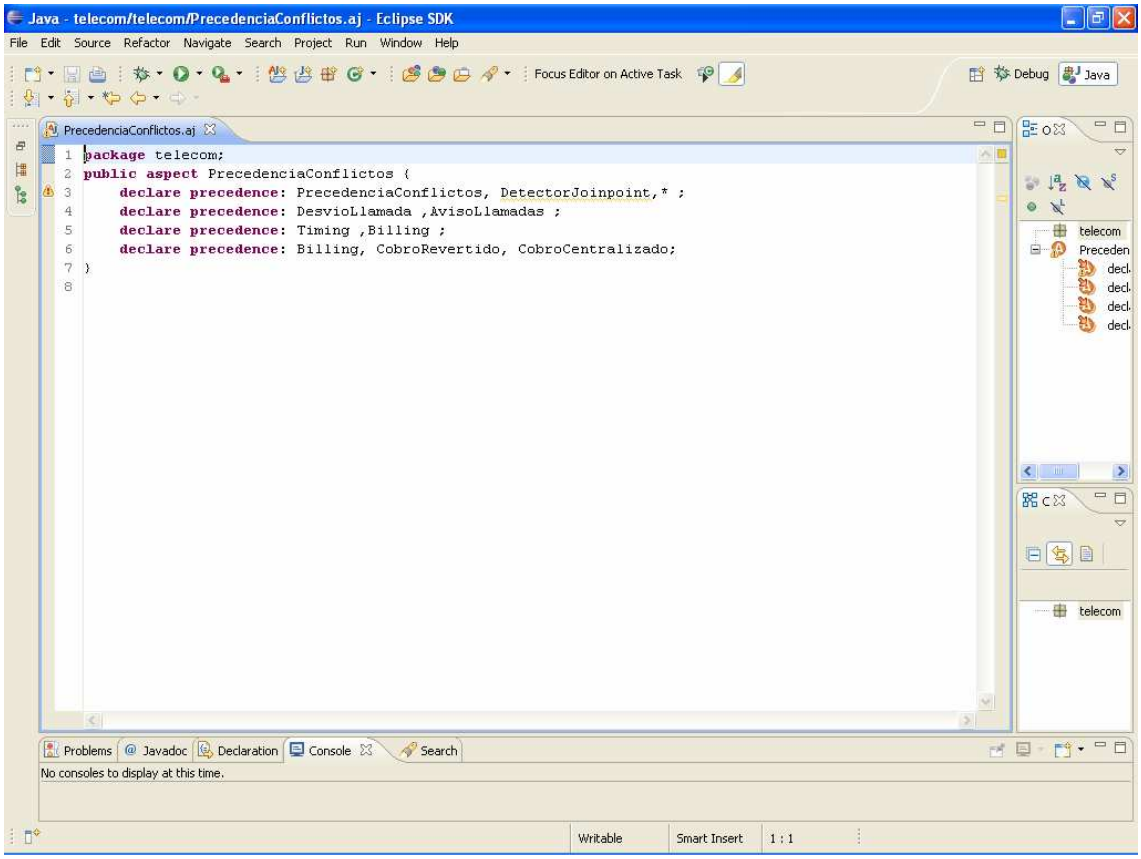
2. Modificar en el aspect DetectorJoinpoint del paquete detectorConflictos la variable 'pruebas ' con el valor a true. De esta forma el detector irá guardando los datos de los joinpoints en la tabla Componentes e irá analizando la información para detectar y solucionar los conflictos detectados.



3. Ejecutar la batería de pruebas creada para la aplicación Telecom, que en la aplicación se encuentra en la clase AbstractSimulation. Antes de la batería, se inicializarán las tablas de Componentes y la tabla de conflictos ConflictosCab mediante la llamada al método iniConflictos de la clase DetectorConflict. Al final de la batería, se ejecutará el método detectarConflictos de la clase DetectorConflict para obtener los conflictos producidos al ejecutar las pruebas.



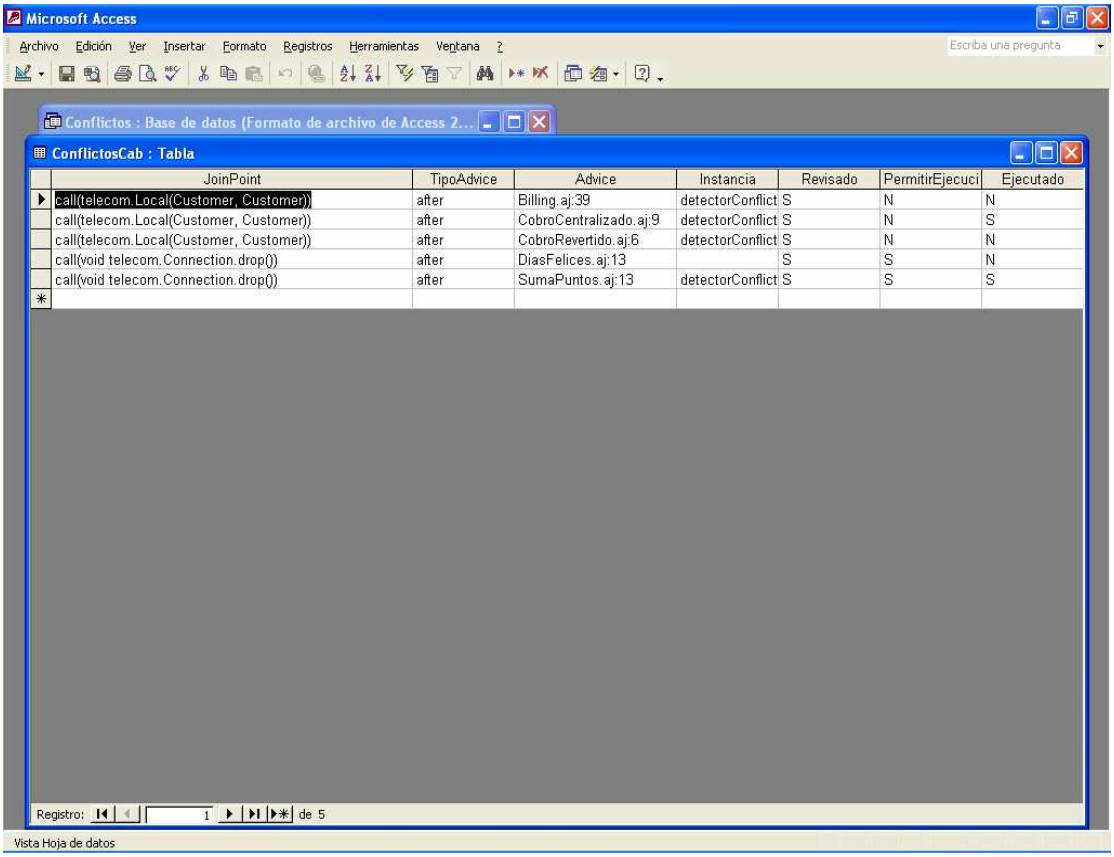
4. El desarrollador/es de la aplicación podrán comprobar los conflictos detectados y solucionados por el detector de conflictos, abriendo el aspect PrecedenciaConflictos generado en la ejecución de las pruebas.



5. El desarrollador/es de la aplicación tendrán que abrir la tabla Access ConflictosCab para ver qué conflictos el sistema no ha sido capaz de solucionar. El desarrollador/es tendrán que comprobar los aspects involucrados para rellenar los campos en la tabla ConflictosCab:

   - Revisado: indicador 'S/N'. El valor 'S' indica que el desarrollador ha revisado el aspect y la información del campo PermitirEjecución puede ser utilizada en la ejecución de la aplicación.

   - PermitirEjecucion: indicador 'S/N'. El valor 'S' indica que si el advice llega a ejecutarse dentro de la aplicación, éste puede ejecutarse aunque otro advice de otro aspect con el que había un conflicto potencial, se haya ejecutado.



6. Una vez que los conflictos han sido solucionados a través de declaraciones de precedencia o mediante órdenes en la tabla ConflictosCab, se procede a modificar en el aspect DetectorJoinpoint del paquete detectorConflictos la variable 'pruebas ' con el valor a false. De esta forma la ejecución se realizará normalmente, actuando de acuerdo a las declaraciones de precedencia y las órdenes, y de esta forma evitar los conflictos entre aspects.

4.3.5 Ejecución Telecom en fase de pruebas

El resultado de la ejecución de la aplicación Telecom en fase de pruebas es (en rojo se muestra la ejecución de los aspects de Telecom):

1. Inicializando componentes
   En este punto se borra el contenido de las tablas Componentes y Precedencias
2. Inicializando Conflictos
   También se borra el contenido de la tabla ConflictosCab
3. Constructor Customer Jim
   se crea el usuario Jim

4. Constructor Customer Mik
   se crea el usuario Mik
5. Constructor Customer Crista
   se crea el usuario Crista
6. jim calls crista...
   el usuario jim hace una llamada al usuario crista
7. Desvio Llamada para Jim de area 650 a area 999
   al realizarse la llamada se lanza el advice correspondiente del aspect DesvioLlamada, cambiando dicho advice el área del usuario que llama al mismo área que la del usuario llamado
8. Atencion, Llamada desde área 999 a area 999
   al realizarse la llamada se lanza el advice correspondiente del aspect AvisoLlamadas que muestra de qué area a qué area se realiza la llamada.
9. [new local connection from telecom.Customer@14a9972 to telecom.Customer@1f630dc]
   se establece la conexión entre los dos usuarios
10. Cobro Centralizado
    al establecerse la conexión se lanza el correspondiente advice del aspect CobroCentralizado. Al ser jim el usuario que llama, el usuario que paga pasa a ser el usuario que actúa como central de pago.
11. En CobroRevertido
    al establecerse la conexión se lanza el correspondiente advice del aspect CobroRevertido. Al ser crista el usuario llamado, el usuario que paga pasa a ser el usuario llamado, es decir, crista.
    Cobro Revertido a Crista
12. Trace: void telecom.Call.pickup()
    el usuario crista accepta la llamada y el aspect Trace recoge este hecho
13. connection completed
    al aceptar la llamada el usuario crista, la conexión pasa a estado completed.
14. Timer started: 1242390949839
    el aspect TimerLog recoge el hecho de que al completarse la conexión un timer empieza a contabilizar el tiempo de duración de la llamada
15. jim hangups...
    el usuario jim termina la llamada
16. connection dropped
    al terminar la llamada se elimina la conexión creada
17. Trace: void telecom.Call.hangup(Customer)
    el aspect Trace visualiza que uno de los usuarios ha colgado la llamada
18. connection dropped
    al terminarse la llamada la conexión pasa al estado dropped

19. Timer stopped: 1242390952042
    al terminarse la conexión se lanza el advice del aspect Timing, el cual para el timer, y así determinar la duración de la llamada.
    Timer passed: 2203
20. Dias Felices: puntos doble
    al terminarse la conexión se lanza el advice extraPuntos del aspect DiasFelices. Este advice sumará cuatro puntos a los puntos que tuviera el usuario que llama.
21. Suma de puntos
    al terminarse la conexión se lanza el advice sumaPuntos del aspect SumaPuntos. Este advice sumará un punto a los puntos que tuviera el usuario que llama.
22. Detectando Conflictos .....
    una vez finalizada la batería de pruebas se ejecuta el procedimiento detectarConflictos de la clase DetectorConflict. Este procedimiento analizará la tabla Componentes para detectar los conflictos.
23. Grabando Precedencias....
    Para los conflictos que el detector puede solucionar, grabará la tabla Precedencias, la cual posteriormente dará lugar al aspect con las declaraciones de precedencia.
24. Fin Grabando Precedencias
25. Fin Detectando Conflictos

4.3.6 Tablas Access

A continuación se muestra un resumen de las tablas Access utilizadas por el detector de conflictos.

Primero se muestra el contenido de la tabla Componentes correspondiente a los registros generados cuando se disparan los aspects AvisoLlamadas y DesvioLlamadas. Estos dos aspects no generaron conflicto potencial, ya que el aspect DesvioLlamada modificaba el atributo Customer.area, mientras que el aspect AvisoLlamadas leía ese mismo atributo.

Aspecto	Inst	F	Joinpoint	Localización	TipAdv	TipCorte	Joinpoint Padre	Conflicto
DesvioLlamada.aj	19bd03e	N	call(telecom.Call(Customer, Customer))	DesvioLlamada.aj:8	before			
DesvioLlamada.aj	3e86d0	N	call(String telecom.Customer.getName())	DesvioLlamada.aj:10	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	19fcc69		String telecom.Customer.name)	Customer.java:122	Before	field-get	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	3e86d0	S	call(String telecom.Customer.getName())	DesvioLlamada.aj:10	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	12f6684	N	call(int telecom.Customer.getAreacode())	DesvioLlamada.aj:10	Before	method-call	call(telecom.Call(Customer, Customer))	

DesvioLlamada.aj	131f71a		int telecom.Customer .areacode)	Customer.java:70	Before	field-get	call(telecom.Call(Cust omer, Customer))	
DesvioLlamada.aj	12f6684	S	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :10	Before	method-call	call(telecom.Call(Cust omer, Customer))	
DesvioLlamada.aj	119c082	N	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :10	Before	method-call	call(telecom.Call(Cust omer, Customer))	
DesvioLlamada.aj	eee36c		int telecom.Customer .areacode)	Customer.java:70	Before	field-get	call(telecom.Call(Cust omer, Customer))	
DesvioLlamada.aj	119c082	S	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :10	Before	method-call	call(telecom.Call(Cust omer, Customer))	
DesvioLlamada.aj	11a698a	N	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :11	Before	method-call	call(telecom.Call(Cust omer, Customer))	
DesvioLlamada.aj	26e431		int telecom.Customer .areacode)	Customer.java:70	Before	field-get	call(telecom.Call(Cust omer, Customer))	
DesvioLlamada.aj	11a698a	S	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :11	Before	method-call	call(telecom.Call(Cust omer, Customer))	

DesvioLlamada.aj	17943a4	N	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :11	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	14fe5c		int telecom.Customer .areacode)	Customer.java:70	Before	field-get	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	17943a4	S	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :11	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	93dee9	N	call(String telecom.Customer .getName())	DesvioLlamada.aj :11	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	df6ccd		String telecom.Customer .name)	Customer.java:12 2	Before	field-get	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	93dee9	S	call(String telecom.Customer .getName())	DesvioLlamada.aj :11	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	f4a24a	N	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :12	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	1a16869		int telecom.Customer .areacode)	Customer.java:70	Before	field-get	call(telecom.Call(Customer, Customer))	

DesvioLlamada.aj	f4a24a	S	call(int telecom.Customer .getAreacode())	DesvioLlamada.aj :12	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	18a992f	N	call(void telecom.Customer .setAreacode(int))	DesvioLlamada.aj :12	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	1fc4bec		int telecom.Customer .areacode)	Customer.java:75	Before	field-set	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	18a992f	S	call(void telecom.Customer .setAreacode(int))	DesvioLlamada.aj :12	Before	method-call	call(telecom.Call(Customer, Customer))	
DesvioLlamada.aj	19bd03e	S	call(telecom.Call(Customer, Customer))	DesvioLlamada.aj :8	Before			
AvisoLlamadas.aj	19189e1	N	call(telecom.Call(Customer, Customer))	AvisoLlamadas.aj: 8	Before		call(telecom.Call(Customer, Customer))	
AvisoLlamadas.aj	1690726	N	call(int telecom.Customer .getAreacode())	AvisoLlamadas.aj: 10	Before	method-call	call(telecom.Call(Customer, Customer))	
AvisoLlamadas.aj	9931f5		int telecom.Customer .areacode)	Customer.java:70	Before	field-get	call(telecom.Call(Customer, Customer))	

AvisoLlamadas.aj	1690726	S	call(int telecom.Customer .getAreacode())	AvisoLlamadas.aj: 10	Before	method-call	call(telecom.Call(Customer, Customer))	
AvisoLlamadas.aj	14b7453	N	call(int telecom.Customer .getAreacode())	AvisoLlamadas.aj: 10	Before	method-call	call(telecom.Call(Customer, Customer))	
AvisoLlamadas.aj	1d5550d		int telecom.Customer .areacode)	Customer.java:70	Before	field-get	call(telecom.Call(Customer, Customer))	
AvisoLlamadas.aj	14b7453	S	call(int telecom.Customer .getAreacode())	AvisoLlamadas.aj: 10	Before	method-call	call(telecom.Call(Customer, Customer))	
AvisoLlamadas.aj	19189e1	S	call(telecom.Call(Customer, Customer))	AvisoLlamadas.aj: 8	Before			

A continuación, mostramos otro resumen de la tabla Componentes correspondiente a los registros generados al dispararse los aspect Billing, CobroCentralizado y CobroRevertido. Estos tres aspects generaron un conflicto potencial debido a que los tres aspects modificaban el atributo Connection.payer.

Aspecto	Inst	F	Joinpoint	Localización	TipAdv	TipCorte	Joinpoint Padre	Conflicto
Billing.aj	9ed927	N	call(telecom.Local(Customer, Customer))	Billing.aj:39	After			

Billing.aj	19616c7		Customer telecom.Connection.payer)	Billing.aj:42	After	field-set	call(telecom.Local(Customer, Customer))	S
Billing.aj	9ed927	S	call(telecom.Local(Customer, Customer))	Billing.aj:39	After			
CobroCentralizado.aj	750159	N	call(telecom.Customer(String, int, int))	CobroCentralizado.aj:7		constructor-call		
CobroCentralizado.aj	1971afc		long telecom.Customer.totalCharge)	Billing.aj:71		field-set	call(telecom.Customer(String, int, int))	
CobroCentralizado.aj	1c39a2d		long telecom.Customer.totalConnectTime)	Timing.aj:27		field-set	call(telecom.Customer(String, int, int))	
CobroCentralizado.aj	13bad12		Vector telecom.Customer.calls)	Customer.java:31		field-set	call(telecom.Customer(String, int, int))	
CobroCentralizado.aj	1632c2d		String telecom.Customer.name)	Customer.java:52		field-set	call(telecom.Customer(String, int, int))	
CobroCentralizado.aj	60420f		int telecom.Customer.areacode)	Customer.java:53		field-set	call(telecom.Customer(String, int, int))	

CobroCentralizado.aj	540408	N	call(void telecom.Customer.setPuntos(int))	Customer.java:54		method-call	call(telecom.Customer(String, int, int))	

CobroCentralizado.aj	1a626f		int telecom.Customer.puntos)	Customer.java:79		field-set	call(telecom.Customer(String, int, int))	
CobroCentralizado.aj	540408	S	call(void telecom.Customer.setPuntos(int))	Customer.java:54		method-call	call(telecom.Customer(String, int, int))	
CobroCentralizado.aj	750159	S	call(telecom.Customer(String, int, int))	CobroCentralizado.aj:7		constructor-call		
CobroCentralizado.aj	1f934ad		Customer telecom.CobroCentralizado.cent)	CobroCentralizado.aj:7		field-set		
CobroCentralizado.aj	f0eed6	N	call(telecom.Local(Customer, Customer))	CobroCentralizado.aj:9	After			
CobroCentralizado.aj	691f36	N	call(String telecom.Customer.getName())	CobroCentralizado.aj:13	After	method-call	call(telecom.Local(Customer, Customer))	
CobroCentralizado.aj	e94e92		String telecom.Customer.name)	Customer.java:122	After	field-get	call(telecom.Local(Customer, Customer))	
CobroCentralizado.aj	691f36	S	call(String telecom.Customer.getName())	CobroCentralizado.aj:13	After	method-call	call(telecom.Local(Customer, Customer))	

CobroCentralizado.aj	fd54d6		Customer telecom.CobroCentralizado.cent)	CobroCentralizado.aj:15	after	field-get	call(telecom.Local(Customer, Customer))	
CobroCentralizado.aj	1174b07		Customer telecom.Connection.payer)	CobroCentralizado.aj:15	After	field-set	call(telecom.Local(Customer, Customer))	S
CobroCentralizado.aj	f0eed6	S	call(telecom.Local(Customer, Customer))	CobroCentralizado.aj:9	After			
CobroRevertido.aj	1d520c4	N	call(telecom.Local(Customer, Customer))	CobroRevertido.aj:6	After			
CobroRevertido.aj	16fd0b7	N	call(String telecom.Customer.getName())	CobroRevertido.aj:10	After	method-call	call(telecom.Local(Customer, Customer))	
CobroRevertido.aj	b753f8		String telecom.Customer.name)	Customer.java:122	After	field-get	call(telecom.Local(Customer, Customer))	
CobroRevertido.aj	16fd0b7	S	call(String telecom.Customer.getName())	CobroRevertido.aj:10	After	method-call	call(telecom.Local(Customer, Customer))	
CobroRevertido.aj	1f436f5	N	call(String telecom.Customer.getName())	CobroRevertido.aj:11	after	method-call	call(telecom.Local(Customer, Customer))	
CobroRevertido.aj	1786e64		String telecom.Customer.name)	Customer.java:122	After	field-get	call(telecom.Local(Customer, Customer))	

CobroRevertido.aj	1f436f5	S	call(String telecom.Customer.getName())	CobroRevertido.aj:11	After	method-call	call(telecom.Local(Customer, Customer))	
CobroRevertido.aj	2808b3		Customer telecom.Connection.payer)	CobroRevertido.aj:12	After	field-set	call(telecom.Local(Customer, Customer))	S
CobroRevertido.aj	1d520c4	S	call(telecom.Local(Customer, Customer))	CobroRevertido.aj:6	After			

A continuación, se muestra el contenido de la tabla de conflictos ConflictosCab, la cual contiene los conflictos detectados en la fase de pruebas:

JoinPoint	TipoAdvice	Advice	Instancia	Revisado	PermitirEjecucion	Ejecutado
call(telecom.Local(Customer, Customer))	after	Billing.aj:39		N		N
call(telecom.Local(Customer, Customer))	after	CobroCentralizado.aj:9		N		N
call(telecom.Local(Customer, Customer))	after	CobroRevertido.aj:6		N		N
call(void telecom.Connection.drop())	after	DiasFelices.aj:13		N		N
call(void telecom.Connection.drop())	after	SumaPuntos.aj:13		N		N

También se muestra el contenido de la tabla Precedencias que contiene información de las precedencias necesarias para evitar ciertos conflictos:

Id	Joinpoint	AspectoPrimero	AspectoSegundo
886	int telecom.Customer.areacode)	DesvioLlamada.aj	AvisoLlamadas.aj
887	long telecom.Timer.stopTime)	Timing.aj	Billing.aj

Después de la fase de pruebas, el desarrollador revisa la tabla de conflictos ConflictosCab. A continuación, modifica el aspect PrecedenciaConflictos y la tabla ConflictosCab para indicar que ante el joinpoint call(telecom.Local(Customer,Customer)), tiene prioridad el advice del aspect CobroCentralizado, y que los otros dos advice solo se ejecutarán en caso de no ejecutarse ningún otro de los advice.

Después de la revisión así quedan la tabla ConflictosCab y el aspect PrecedenciasConflictos:

JoinPoint	TipoAdvice	Advice	Instancia	Revisado	PermitirEjecucion	Ejecutado
call(telecom.Local(Customer, Customer))	after	Billing.aj:39		S		N
call(telecom.Local(Customer, Customer))	after	CobroCentralizado.aj:9		S		N
call(telecom.Local(Customer, Customer))	after	CobroRevertido.aj:6		S		N
call(void telecom.Connection.drop())	after	DiasFelices.aj:13		S		S
call(void telecom.Connection.drop())	after	SumaPuntos.aj:13		S		S

El detector generó las tres primeras sentencias de precedencia, mientras que la última fue añadida por el desarrollador para solucionar el conflicto reflejado en la tabla de conflictos ConflictosCab entre los aspects Billing,

CobroRevertido y CobroCentralizado. Con esta última sentencia se indica que tendrá prioridad para ejecutarse el advice del aspect CobroCentralizado, a continuación el advice del aspect CobroRevertido, y por último el del aspect Billing (al tratarse de advice de tipo 'after', el aspecto con mayor precedencia ejecutará su advice after sobre un join point después que otro de menor precedencia).

```
package telecom;
public aspect PrecedenciaConflictos {
	declare precedence: PrecedenciaConflictos, DetectorJoinpoint,* ;
	declare precedence: DesvioLlamada ,AvisoLlamadas ;
	declare precedence: Timing ,Billing ;
	declare precedence: Billing, CobroRevertido, CobroCentralizado;
}
```

4.3.7 Ejecución Telecom en fase de ejecución

El resultado de la ejecución de la aplicación Telecom en fase de ejecución es la siguiente (en rojo se muestra la ejecución de los aspects de Telecom):

1. Constructor Customer Jim
   se crea el usuario Jim
2. Constructor Customer Mik
   se crea el usuario Mik
3. Constructor Customer Crista
   se crea el usuario Crista
4. jim calls crista
   el usuario jim llama al usuario crista
5. Desvio Llamada para Jim de area 650 a area 999
   al realizarse la llamada se lanza el advice correspondiente del aspect DesvioLlamada, cambiando dicho advice el área del usuario que llama al mismo que el del usuario llamado. Este advice se ejecuta antes que el advice del aspect AvisoLlamadas por la declaración de precedencia que el detector generó en la fase de pruebas, y que queda reflejado en el aspect PrecedenciasConflictos.
6. Atencion, Llamada desde área 999 a area 999
   al realizarse la llamada se lanza el advice correspondiente del aspect AvisoLlamadas que muestra de qué area a qué area se realiza la llamada. Por lo dicho en el punto anterior, este advice se lanza posteriormente al advice del aspect DesvioLlamada.
7. [new local connection from telecom.Customer@d1e604 to telecom.Customer@e4f972]
   se establece la conexión entre los dos usuarios
8. Cobro Centralizado
   al establecerse la conexión se lanza el correspondiente advice del aspect CobroCentralizado. Al ser jim el usuario que llama, el usuario que paga pasa a ser el usuario que actúa como central de pago.
   Este advice se ejecuta primero al estar indicado en la declaración de precedencia del aspect creado por el desarrollador Precedencias. Además, se ejecuta porque es el primer advice que se ejecuta del conflicto que fue detectado en la fase de pruebas.
9. Hay un conflicto - no ejecutar CobroRevertido.aj:6
   al establecerse la conexión se lanza el correspondiente advice del aspect CobroRevertido, pero no se ejecuta al haberse ejecutado ya otro advice que formaba parte de un conflicto

en el que ambos forman parte y estar indicado en la tabla ConflictosCab que no se ejecute (PermitirEjecucion = "N").

10. Hay un conflicto - no ejecutar Billing.aj:39
    al establecerse la conexión se lanza el correspondiente advice del aspect Billing, pero no se ejecuta al haberse ejecutado ya otro advice que formaba parte de un conflicto en el que ambos forman parte y estar indicado en la tabla ConflictosCab que no se ejecute (PermitirEjecucion = "N").
11. crista accepts call from jim...
    el usuario crista accepta la llamada de jim
12. Trace: void telecom.Call.pickup()
    el usuario crista accepta la llamada y el aspect Trace recoge este hecho
13. Timer started: 1242756802204
    el aspect TimerLog recoge el hecho de que al completarse la conexión un timer empieza a contabilizar el tiempo de duración de la llamada
14. jim hangups...
    el usuario jim termina la llamada
15. connection dropped
    al terminar la llamada se elimina la conexión creada
16. Trace: void telecom.Call.hangup(Customer)
    el usuario termina la llamada y el aspect Trace avisa
17. Timer stopped: 1242756802805
    al terminarse la conexión se lanza el advice del aspect Timing, el cual para el timer, y así determinar la duración de la llamada.
    Timer passed: 601
18. Dias Felices: puntos doble
    al terminarse la conexión se lanza el advice extraPuntos del aspect DiasFelices. Este advice se ejecuta pese a formar parte de un conflicto reflejado en la tabla ConflictosCab, ya que el desarrollador decidió permitir la ejecución tanto del advice del aspect DiasFelices como del advice del aspect SumaPuntos (PermitirEjecucion="S").
19. Suma de puntos

5 CONCLUSIONES Y TRABAJOS FUTUROS

La POA es un nuevo paradigma, aún poco consolidado pero que ofrece muchas posibilidades a los desarrolladores de software, lo cual está haciendo que su popularidad esté creciendo. La POA ofrece una buena solución para el problema de modelar los concerns que afectan a diferentes partes del sistema, y que con la POO se encontraban dispersos en múltiples clases.

La POA no sustituye a la POO, sino que la complementa, coexistiendo las clases y los aspects. Las clases contienen la lógica de la funcionalidad básica del sistema, mientras que los aspectos contienen la lógica de los concerns.

Las ventajas de la POA son múltiples:

- Separación completa de concerns.
- Diseño más modular.
- Mantenimiento y evolución del sistema más fácil.
- Reducción de costes.
- Da la posibilidad de retrasar decisiones sobre los concerns.

Pero la POA también tiene inconvenientes:

- Nuevo paradigma aún poco utilizado entre los desarrolladores de software.
- Escasez de herramientas que faciliten el uso de la POA.
- Conflictos entre aspects que afectan a un mismo concern.

En el presente proyecto se han estudiado los conflictos entre aspects y se ha desarrollado una herramienta para la detección y resolución de algunos de estos conflictos utilizando la propia POA. En concreto, se han tratado los conflicos que afectan a la lectura y escritura de los atributos de las clases de un sistema.

Entre los futuros trabajos que se podrían hacer a partir de este proyecto estarían:

- Investigar cómo se podría mejorar el detector de conflictos para que se pueda utilizar en aplicaciones con múltiples hilos. El presente proyecto asegura que el detector funciona en aplicaciones secuenciales, pero no en aplicaciones con múltiples hilos.

- Investigar si nuevas versiones de AspectJ permiten eliminar la restricción de tener que definir una variable de tipo JoinPoint para que funcione el joinpoint adviceexecution utilizado para la detección de conflictos.

- Ampliar la detección de conflictos a otro tipo de conflictos que no sean los producidos al intentar acceder en modo escritura. Por ejemplo, conflictos basados en funcionalidad similar sobre un mismo joinpoint.

Además de mejorar el presente proyecto hay otras líneas que podrían seguirse para avanzar en la detección de conflictos y hacer más fácil el uso de la POA. Estudiar las nuevas versiones que aparezcan de las herramientas mencionadas en el apartado 3.9, o de nuevas herramientas que hagan más fácil el desarrollo de aplicaciones POA, ayudando también a crear aplicaciones libres de conflictos.

6 ANEXOS

6.1 Instalación de Eclipse y AJDT

- Obtener Eclipse: El IDE Eclipse puede obtenerse bajándolo directamente del sitio web oficial del Proyecto Eclipse. Al estar Eclipse escrito en Java , es necesario para su ejecución que exista un JRE (Java Runtime Environment) instalado previamente en el sistema. Para la instalación de Eclipse, basta con descomprimir el archivo descargado en el directorio conveniente.

- Descargar e instalar AJDT 1.6.1: Obtener el archivo que contiene los plug-ins del sitio org.eclipse.ajdt/download.html. Hay que seleccionar la versión correcta de acuerdo a la versión de Eclipse. En nuestro caso, la versión 1.6.1 para la versión 2.3 de Eclipse. A continuación habrá que descomprimir el archivo que se ha bajado en el directorio de instalación de Eclipse.

- También se puede instalar AJDT añadiendo un nuevo site en la opción "Software Updates" dentro de Eclipse. En nuestro caso, se añadió el sitio download.eclipse.org/tools/ajdt/34/update.

En estos momentos, ya se puede utilizar Eclipse.

6.2 Crear un nuevo proyecto que utiliza aspects en Eclipse:

- Hacer clic en File/New/AspectJ Project en la barra de herramientas.

- Se abre una nueva ventana donde se escribirá el nombre del proyecto a crear. Hacer clic en Next.

- Se abre una nueva ventana. En la etiqueta Libraries debe aparecer la versión del JRE System Library (en nuestro caso jre6). Si no aparece en la lista, se debe agregar haciendo clic en el botón Add Library. Por último, haga clic en Finish.

6.3 Compilación y Ejecución de un proyecto de Eclipse

Para compilar un proyecto que utiliza aspects se puede seleccionar Build - Project en la opción Project de la barra de herramientas. También puede seleccionarse la opción Build Automatically para que se compile automaticamente.

Si queremos ejecutar el proyecto habrá que seleccionar Run Configurations en la opción Run de la barra de herramientas. En el campo Project se escribirá el nombre del proyecto a ejecutar y en el campo Main class debe especificarse la clase del proyecto que contiene el main. Por último, se hará clic en Run.

7 REFERENCIAS

[1] N Degrees of Separation: Multi-Dimensional Separation of Concerns. Tarr. P., Ossher H., Harrison W., Sutton Jr. S. M. In Proceedings of the 21st International Conference on Software Engineering (ICSE'99), pp. 107-119, 1999.

[2] Administración de Conflictos entre Aspectos en AspectJ. Sandra Casas, Claudia Marcos, Verónico Vanoli, Hector Reinaga, Luis Sierpe, Jane Pryor y Claudio Saldivia. In Proceedings of the Six Argentine Symposium on Software Engineering (ASSE´2005) ISSN 1666 1087, 34 JAIIO (Jornadas Argentinas de Informática e Investigación Operativa) ISSN 1666 1141, pp 111-125. Rosario Agosto 2005.

[3] I want my AOP (Part 1). Ramnivas Laddad. Java World, ITworld.com, 2002.

[4] Aspect-Oriented Programming is Quantification and Obliviousness. Robert E. Filman, Daniel P. Friedman. RIACS Technical Report 01.12. May 2001.

[5] www.parc.com. Página de PARC. Noviembre 2007.

[6] http://www.ccs.neu.edu/research/demeter. Página del grupo de Demeter. Noviembre 2007.

[7] Programación Orientada a Aspectos. Análisis del paradigma. Tesis de Licenciatura. Fernando Asteasuain. Bernardo Ezequiel Contreras. Octubre 2002.

[8] AspectJ Cookbook. O'Reilly (2005). Russ Miles.

[9] Reasoning About Semantic Conflicts Between Aspects. Pascal Durr, Tom Staijen, Lodewijk Bergmans, Mehmet Aksit. In: EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software.

[10] Clasificación y Resolución de Conflictos entre Aspectos – Sandra Casas, Héctor Reinaga, Luis Sierpe, Verónica Vanoli, Claudio Saldivia, Jane Pryor. Anales del VII Workshop de investigadores en Ciencias de la Computación. WICC 2005 Agosto de 2005 I.S.B.N.:950-665-337-2 Coordinación de Comunicación Institucional

Equipo de Producción Editorial Coordinador: Lic. Miguel A. Tréspidi. Registro: Daniel Ferniot.

[11] Conflictos entre Aspectos en Etapas del Desarrollo de Software. . Sandra Casas, Verónica Vanoli, Claudia Marcos y Eugenia Márquez. IX Workshop de Investigadores en Ciencias de la Computación (WICC07). Pp 360 - 364 Facultad de Ingeniería. Universidad Nacional de la Patagonia San Juan Bosco. Trelew. Chubut. Argentina. 03 y 04 de Mayo. ISBN Nº: 978-950-763-075-0.

[12] Alpheus, una herramienta para la construcción de aplicaciones reflexivas

[13] ASTOR: Un Prototipo para la Administración de Conflictos en AspectJ. En co-autoria con Sandra Casas, Verónica Vanoli, Hector Reinaga, Luis Sierpe, Jane Pryor, Claudio Valdivia. XIII Jornadas Chilenas de Computación. Valparaiso Chile. Noviembre 2005. pp: 1-10.

[14] A.Tripathi, E.Berge and M.Askit. An Implementation of the Object-Oriented Concurrent Programming Language Sina, Software: Practice and Experience, Vol. 19 (3), March 1989.

[15] Página de AspectJ: http://eclipse.org/aspectj/. Marzo 2008.

[16] Página de AspectC++: http://www.aspectc.org/. Abril 2008.

[17] Página de Eos: http://www.cs.virginia.edu/eos/. Abril 2008.

[18] Página de phpAspect: http://phpaspect.org/. Abril 2008.

[19] Página de AspectC++: . Abril 2008.

[20] F.Tessier, M. Badri, L. Badri. A Model-Based Detection of Conflicts Between Crosscutting Concerns: Towards a Formal Approach. In: International Workshop on Aspect-Oriented Software Development. (2004)

[21] M. Rinard, A. Salcianu, S. Bugrara. A Classification System and Analysis for Aspect-Oriented Programs. Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, October 31-November 06, 2004, Newport Beach, CA, USA.

[22] Paolo Falcarin, Marco Torchiano. Automated Reasoning on Aspects Interactions. ASE 2006: 313-316.

[23] A.J. Bernstein. Program analysis for parallel processing. IEEE Transactions on Electronic Computers, EC-15(5) (1966) 757—762.

[24] Página del proyecto Jess. http://herzberg.ca.sandia.gov/jess/. Junio 2008.

[25] André Restivo, Ademar Aguiar. Towards Detecting and Solving Aspect Conflicts and Interferences Using Unit Tests. Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies, p.7-es, March 12-16, 2007, Vancouver, British Columbia, Canada.